Practical DevOps

DevOps 实践

驭DevOps之力强化技术栈并优化IT运行

[瑞典]Joakim Verona 著
高清华 马博文 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Practical DevOps

DevOps实践

[瑞典]Joakim Verona 著

高清华 马博文 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

《DevOps 实践》介绍了 DevOps 的起源和概览，并通过一个贯穿全书的例子，从架构开始，到代码的存储、构建、测试、部署、监控，直至流程的跟踪，推荐了许多可用的工具和可行的示范，是一本 DevOps 实践方面不可多得的参考书籍。

本书面向愿意承担更大责任的开发人员和系统管理员，也很适合愿意更好地支持开发人员的运维人员。无须任何 DevOps 知识即可快速上手！

版权贸易合同登记号　图字：01-2016-3999

图书在版编目（CIP）数据

DevOps 实践 /（瑞典）约阿基姆 • 维罗纳（Joakim Verona）著；高清华，马博文译.
北京：电子工业出版社，2016.10
书名原文：Practical DevOps
ISBN 978-7-121-29812-7

Ⅰ.①D… Ⅱ.①约… ②高… ③马… Ⅲ.①虚拟处理机 Ⅳ.①TP338

中国版本图书馆 CIP 数据核字(2016)第 207459 号

责任编辑：张春雨
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：13.5　字数：302.4 千字
版　　次：2016 年 10 月第 1 版
印　　次：2016 年 10 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

译者序

什么是 DevOps？我的前同事李光磊将其译为开发自运维，他还写了篇很有意思的博客来说明：http://liguanglei.name/blogs/2015/04/22/devops-chinese-name/。这个将开发和运维结合起来的词，代表了一种文化，那就是大家共同协作。狭义上的大家，指的是开发和运维，广义上，指的是所有软件生命周期里参与的角色。

“共同协作”是个富有正能量的词。感觉上，随便往哪儿一套都是正确的。那为什么要在 DevOps 里着重强调呢？DevOps 到底解决了什么问题？归根结底，就是提高产品质量。爱思考的你，可能心里已经有千万个提高产品质量的方案从脑海里呼啸而过——代码审查、自动化测试、持续集成、代码质量管理工具、程序员鼓励师……对对对，这些方案都能在某种程度上解决一些层次的问题。但是，产品质量的根源在哪儿呢？在于人。如果开发者对自己要做的事情不负责，甚至压根儿不知道后果，怎么能指望这样的人能够生产出来高质量的代码呢？举个例子：作为开发者，我知道自己写的代码会由测试部门来进一步测试，在有进度压力的时候，我就会更倾向于去想：“那就先这么凑合着吧，反正有问题 QA 们会说的”。如果我不知道部署和维护产品是怎么一回事，我就不会主动地在产品里写上日志的代码。对于运维人员来说，由于处于软件生命周期的下游，相信对类似的场景感触更甚。DevOps 能够做到的事，就是让人有这个意识：需要对产品的质量负责。DevOps 能够提供一个平台或机制，让我能够从中找到所需的资源。

“共同协作”也是个虚无缥缈的词。它应该如何落地呢？这就是本书想要给读者们带来的内容。在实践上，从架构开始，到代码的存储、构建、测试、部署、监控，直至流程的跟踪，本书推荐了许多可用的工具和练习，确实无愧于《DevOps 实践》之名。细读全书可以对其有一个全局的概览并充实自己的 DevOps 工具箱；而在实际场景中再查阅本书，将其当作一本各种技术的快速参考手册也不失为明智之举。本书的许多实例通过 Docker 启动，在紧随潮流技术的同时也简化了练习步骤，值得花些时间试试。在企业里，使用自动化和持续交付来提高代码部署频率、降低代码上线间隔。这样的指标是比较容易统计的，在让管理人员满意的同时，也能减少开发和运维的痛苦。只有让各角色都真切地感受到实

惠，大家才会更加愿意从心底接受并积极参与到这一过程中。

“共同协作”是个看上去很美的词。为什么大家还不赶紧拥抱它？因为它的成本可能还挺高的。大型企业在管理上，通常权责分明，从而导致每个角色的成员都不愿意轻易踏足其他领域；流程烦琐，导致一个小小的改进也需要漫长的批复；安全性要求高，引发各种违规，进一步导致没有和其他人分享的意愿；员工操作权限管理精密，上不了网、下不了包、开不了虚拟机……这些问题，虽然不至于疾在骨髓，但起码也在肠胃了。而且，自动化测试、部署流水线等都需要比较高的成本。在看见收益和认清自己之前，可能大多数人也会像蔡桓公那样默认拒绝吧：“医之好治不病以为功”。成本最低的时候，可能就是开始写第一行产品代码的时候。话虽如此，任何时候都是实现 DevOps 的最佳时机，因为随企业的扩大和代码库的膨胀，成本一定是越来越高的。另外，完全地追求技术上的卓越而忽视成本也不是 DevOps 的推荐做法。读者们在阅读时，也会看到 DevOps 在一些状况下采取的权衡方案。

你希望在一个大家敞开心胸、相互拥抱的环境里共同协作以打造最好的产品，还是守着自己的一亩三分地，与人争辩这是谁的责任，抱怨人们冷漠的同时拒绝其他人的“与你无关”的要求？从本书开始，应用自己获得的知识，并尝试改造这个世界吧！

关于作者

Joakim Verona 是一位擅长持续交付和 DevOps 的咨询师。自 1994 年以来，在系统开发的所有方面他都曾工作过。他积极地在诸如 web 系统、多媒体系统和软硬件混合系统等复杂的多层系统上做出了领导实践者的贡献。自 2004 年以来，他广泛的技能兴趣把他导向了新兴的 DevOps 领域。

Joakim 在林雪平理工学院完成了计算机科学的硕士学位。他也曾作为咨询师工作在各种各样的工业领域中，例如银行和财务、电信、工程、印刷和排版，还有游戏开发。他还对敏捷领域感兴趣，是一位 Scrum 认证的敏捷教练、Scrum 产品负责人并拥有 Java 认证。

我想要谢谢我的妻子，Marie，在写这本书的时候她就是灵感的源泉。我也想谢谢过去数年曾经一起工作过的所有人和公司，让我可以工作在我喜欢的事情上。

关于审稿人

Per Hedman 是一个富有激情的开发者，他也是一个 DevOps 和持续交付的强烈倡导者，他相信应该让开发者对他们自己所写的代码负起责任。

他是一名软件咨询师，在 21 世纪初曾经是一位开发和运维人员。

特别感谢我的妻子和两个女儿，她们让我开心微笑。

Max Manders 是一名运维工程师，在 FanDuel 工作，这家公司是在线 DFS（Daily Fantasy Sports）游戏的领跑者。Max 先前在 Cloudreach 的运营中心工作，这家公司是 AWS Premier 咨询合作伙伴。Max 充分发挥了他的经验和技巧来促进 DevOps 的发展，他也致力于掌握 Ruby 并通过 Chef 和 Puppet 来倡导基础设施及代码。

Max 是 Whisky Web 的共同创立者和组织者，这是一个苏格兰 web 开发和运维社区的大会。当他不写代码或者是研究最新、最棒的监控和运维工具的时候，Max 喜欢威士忌、演奏爵士乐和长号。Max 和他的妻子 Jo 还有他们的猫咪 Ziggy 和 Maggie 生活在爱丁堡。

目录

前言

DevOps 领域在近年来变得流行而普遍。它是那么的流行，以至于很容易忘记在 2008 年以前，当 Patrick Debois 组织起第一个 DevOps 之日大会时，几乎没人曾经听说过该词。

由开发（developers）和运维（operations）组成的 DevOps 这个词，到底意味着什么？为什么它能造成如此巨大的狂热？

本书的任务就是回答这个看起来很简单的问题。

简短的答案就是：DevOps 旨在将不同的社区，比如开发和运维社区，联合起来变成一个更有效率的整体。

这也反映在本书中。它探索了许多在 DevOps 工作中有用的工具，还有那些更加凝聚人们的工具，这些工具比起那些在人之间划清边界的工具来说更令人喜爱。我们用来进行软件开发的流程也是工具，所以将 DevOsp 相关的不同敏捷流派的各个方面包含进来也是很自然的事。

本书也希望做到像标题说的那样，注重实战。让我们在 DevOps 之路上开始旅程吧！

本书主要内容

第 1 章，DevOps 和持续交付简介，涉及了 DevOps 的背景，并介绍它是怎样融入到敏捷开发的广袤世界的。

第 2 章，洞察全局，它会帮助你了解 DevOps 使用的多个系统如何协同工作，组成一个大整体。

第 3 章，DevOps 如何影响架构，描述了软件架构的各个方面，以及当我们以 DevOps 的视角工作时它对我们的意义。

第 4 章，一切皆代码，解释了如何实现一切皆代码。而且，你需要一个地方来存储代码，这个地方就是组织里的源代码管理系统。

第 5 章，构建代码，解释了为何需要系统来构建代码，介绍了这些系统。

第 6 章，测试代码，展示了如果需要及早发布或者经常性发布代码，我们就得对代码的质量有信心。因此我们需要自动化回归测试。

第 7 章，部署代码，展示了当完成了代码的构建和测试，你需要将其部署到服务器上，这样客户就能使用新部署的特性了。

第 8 章，监控代码，涵盖了代码如何通过选择的部署方案来安全地部署到服务器上。你需要监护着它以使其正常工作。

第 9 章，问题跟踪，介绍了处理组织内开发流程的系统，例如问题跟踪软件。在实现敏捷流程时，这样的系统是很重要的帮手。

第 10 章，物联网和 DevOps，描述了 DevOps 如何在物联网的新兴领域帮助我们。

本书的使用要求

本书包含了许多实用例子。为了融会贯通这些例子，你需要一台机器，最好是基于 GNU/Linux 的操作系统，例如 Fedora。

本书的读者

本书面向那些想要承担更大责任，并了解基础设施如何做到构建现代企业的开发者。本书也面向那些想要更好地支持开发者的运维人员。自动化测试的技术人员也是本书的目标受众。

本书主要是包含了许多实例的技术文档，适合那些想要学习实现具体工作代码的人员。尽管如此，前两章的实践性并不强。它们交代了有助于了解其余章节的背景和概览。

约定

在本书中，你将会发现不同的信息类型使用不同的文本样式来区别。这里列出了一些范例和解释：

文本中的代码、数据库表名、文件夹名、文件名、文件扩展名、路径、伪 URL、用户输入和 Twitter 标签以下列形式展示：“在你的本地安装 `git-review`”。

代码段如下所示：

```
private int positiveValue;
void setPositiveValue(int x){
  this.positiveValue=x;
}

int getPositiveValue(){
  return positiveValue;
}
```

命令行的输入输出如下所示：

```
docker run -d -p 4444:4444 --name selenium-hub selenium/hub
```

新术语和**关键词**粗体显示。你在屏幕上看到的词，例如在菜单或者是对话框里，在文本中看起来像是这样：“我们可以通过**修改**按钮改变状态。”

警告或是注意事项像这样展示。

要诀和技巧是这样的。

下载示例代码

你可以从 `http://www.broadview.com.cn` 下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

1

DevOps 和持续交付简介

欢迎来到《DevOps 实践》!

本书的第 1 章将会讨论 DevOps 的背景，并介绍它是怎样融入到敏捷开发的广袤世界的。

关于 DevOps，很重要的一点是如何向身边的同事们解释它到底是什么东西。

如果能让大家更快地登上 DevOps 列车，你就能更快地尝试到真正的技术实践!

在这一章，我们将会涉及到以下话题：

- DevOps 简介。
- 多快才算快?
- 敏捷之轮。
- 敏捷不只是形式。
- DevOps 和 ITIL（信息技术基础架构库）。

DevOps 简介

在定义上，DevOps 是一个涵盖着几条线的领域。它既非常实用又贴近实践。但与此同时，你需要了解的不仅有技术背景，还有非技术的文化方面。本书涵盖了 DevOps 最佳实践的动手和软技能两个部分。

DevOps 由开发（developments）和运维（operations）两个单词组成。这个双关语已经

揭示了 DevOps 的本意，那就是鼓励不同的软件开发部门共同协作。

DevOps 这个词的起源和 DevOps 运动的早期还是很清晰的：Patrick Debois 是一名在 IT 行业的许多领域里很有经验的软件开发工程师兼顾问。他本人对于开发和运维之间的对立感到相当不爽。他试图在会议中引起大家对这个问题的兴趣，但是一开始并没有什么效果。

在 2009 年，O'Reilly Velocity 大会上有个深得好评的演讲："每日至少十次部署：开发和运维在 Flickr 的合作"。Patrick 随即决定在比利时根特市组织一场名为 DevOps 之日的活动。这次，感兴趣的人变多了，这场大会获得了成功。"DevOps 之日"这个名字引起了共鸣，而这场大会也延续了下来。在 Twitter 和大多数论坛里，DevOps 之日被简称为 DevOps。

DevOps 运动的根源写在了敏捷软件开发原则里。在 2001 年，有些人想要改进一成不变的软件开发模式，并寻找新的工作方法，他们编写了敏捷宣言。下面是敏捷宣言里被奉为经典的摘录，可以在 `http://agilemanifesto.org/` 上阅读：

> "个体和互动 高于 流程和工具
> 工作的软件 高于 详尽的文档
> 客户合作 高于 合同谈判
> 响应变化 高于 遵循计划
> 也就是说，尽管右项有其价值，我们更重视左项的价值。"

由此可见，DevOps 可以说是与第一条原则密切相关的——"个体和互动 高于 流程和工具。"

显然这能够给工作带来好处——那为什么我们还要强调这么明显的事呢？如果你在大型企业里工作过，你就会知道事实上经常是反着来的。哪怕是看起来没什么障碍的小企业，里面的各个部门也很容易就筑起高墙。

DevOps，想要强调个体和互动是非常重要的，并且这个技术很可能有助于拆除企业里的部门墙。看起来可能有点儿反直觉，因为第一条原则更青睐于交互而不是工具。但是我认为使用任何工具都能起到多种效果。只要工具用得适当，就能帮我们得到所有想要在敏捷中获得的东西。

举个非常简单的例子，一个选择系统过去经常有缺陷。通常，开发团队和测试团队会用不同的系统来处理任务和缺陷。这样的事不仅在团队中导致了不必要的摩擦，并且把本应一起工作的双方隔离开了。而运维团队很可能又会用第三种系统来处理服务器的部署

请求。

另一方面，有 DevOps 观念的工程师，会立即意识到所有的三个系统都是相似的工作流程。三个团队里的每个人应该都有使用一个相同系统的可能性，也许只需要为不同的角色展示不同的界面就可以了。因为三个系统变成了一个，所以会带来减少维护成本的长期利益。

DevOps 的另一个核心目标是自动化和持续交付。简单来说，自动化一切可重复的乏味的工作，把更多时间留给人与人之间的交流，这才能产生真实的价值。

多快才算快？

DevOps 化的转变必须要快。在高层次上，我们需要考虑抢占市场；在低层次上，我们需要盯紧任务。这也是持续交付运动的思路。

就像敏捷的很多东西一样，DevOps 和持续交付虽然有许多概念名称不同，但是它们都有着相同的含义。它们只不过是一枚硬币的正反面而已，在概念上并没有什么争议。

DevOps 工程师致力于让公司的流程更快、更有效，并且更可靠。只要有可能，就取代那些容易出错的重复性人力劳动。

尽管如此，在 DevOps 实践过程中很容易忘记最终目标。要是干得太慢，对任何人来说都没什么用。相反，我们必须把交付增加的商业价值牢记在心。

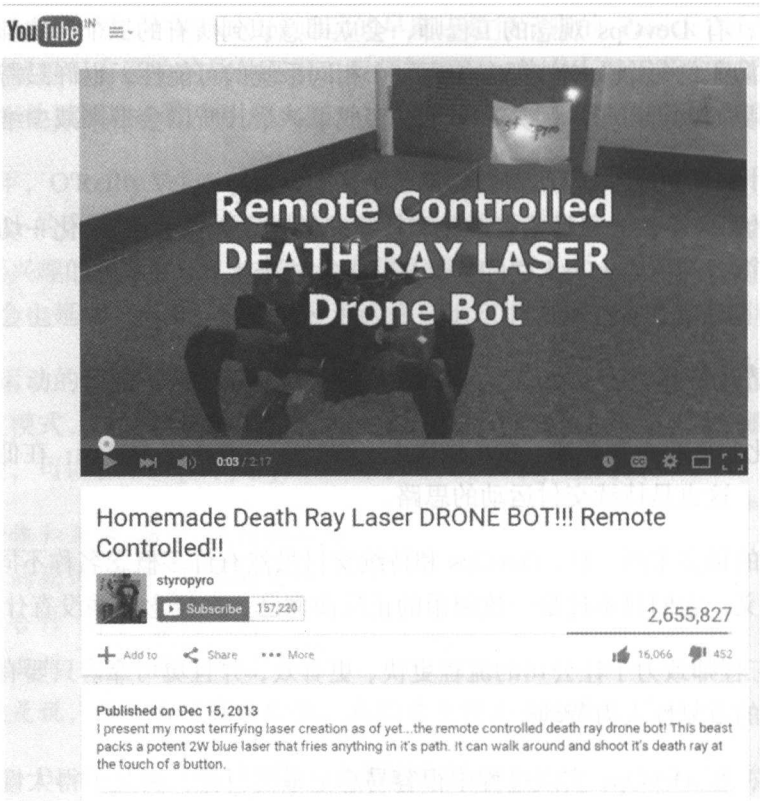
例如，增进企业内部各角色的交流就有很明显的价值。你的产品负责人可能想了解开发的进度并渴望能够先睹为快。这样的话，在测试环境上做到快速并有效地增量交付，将会非常有帮助。像产品负责人这样的利益相关者，当然还有质量保证团队，都能够在测试环境上跟上开发的节奏。

另一种观点是这样的：如果你曾经感觉到自己因为不必要的等待而注意力不集中，那么你的流程或工具有问题。如果你在程序编译时看机器人打气球的视频，那么你的编译时间太长了！

在团队等待部署这件事上也一样。当然了，整个团队白白呆着比一个人开销大多了。

机器人打气球的视频很有意思，不过开发软件也很让人兴奋！我们应该通过消除不必

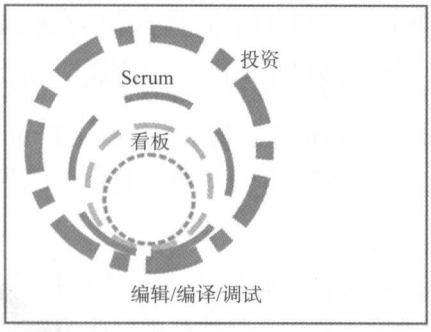
要的开销来关注创造潜能。



死光机器人 vs 团队生产率

敏捷之轮

敏捷开发中有几个不同的周期，从投资级的周期，到 Scrum 和看板周期，再到持续集成周期。根据你使用的敏捷框架的不同，工作的侧重点也都有些许不同。看板所强调的 24 小时周期在运营团队里比较流行。而采用 Scrum 敏捷流程的开发团队，Scrum 周期通常是 2 到 4 周。在大规模敏捷框架（Scaled Agile Framework）中，长周期也非常普遍，称为**项目增量**（Program Increments），可以持续数个 Scrum Sprint 周期。



敏捷之轮

DevOps 要能够支持这些周期更好地运转。在 DevOps 的思想下这再正常不过了：在敏捷的企业中跨部门协作。

DevOps 在短周期中能带来明显的实质效益，而这些效益会让长周期变得更有效率。俗话说不积跬步，无以至千里；不积小流，无以成江海。

下面是一些敏捷周期可以从 DevOps 获益的例子：

- DevOps 工程师维护的部署系统，让 Scrum 周期最后的交付更快、更有效。而交付每 2 到 4 周就会周期性地发生。
  在经常手动部署的企业里，部署一次可能需要好几天。像这样在部署上没有效率的企业可以从 DevOps 观念中获益良多。
- 看板的周期是 24 个小时，所以很显然如果我们想要在看板方法上取得成功，部署的周期需要快得多。
  一旦代码提交到代码库，取决于变更集的大小，一个设计良好的持续交付流水线大约只需要几分钟就可以把提交的代码部署到生产环境。

敏捷不只是形式

由于在量子物理上的成就，Richard Feynman 在 1965 年获得了诺贝尔奖。他注意到科学家中的一个普遍现象，那就是他们覆盖了科学的全部边界，可是就偏偏漏掉了一些关键性因素。他把这种现象称为“草包族科学（cargo cult science）”。这来源于美拉尼西亚南部海岛上的草包族（cargo cult）。草包族这个说法是在第二次世界大战时，因岛上的土著居民

观察到飞机给他们送货而产生的——战争结束之后，货物就不再送来了。土著们便开始模仿以前观察到的美军的行为，比如修建模拟机场，幻想飞机能因此再次回来。

草包族的一架敏捷飞机

如果我们只是早上站个会，喝点咖啡，聊聊天气，这并不是敏捷或者面向 DevOps 的方式。如果我们的 Puppet 只有运维团队才知道怎么用，这也不是 DevOps 流水线。

我们是否正在做正确的事？是否还在正确的路上？时刻关注我们的目标并经常问自己，是件非常重要的事情。这是敏捷思考的中心。然而，在实践中显然非常困难。很容易以草包族的方式而告终。

每当构建部署流水线的时候，举个例子，留神我们为什么要在第一时间构建它们。最终目标是让人们可以更快、更容易地和新系统交互。它帮助不同角色的人们更有效率地沟通，而不用改变太多。

另一方面，如果我们构建的流水线只是帮助了一个组的人们达成他们的目标，比方说

运维组的人，那么对于达成最终目标来说，我们已经打了一场败仗。

虽然没有科学根据，但值得记住的是敏捷周期，比如 Scrum 的 sprint 周期，一般都会有办法来应对这种状况。Scrum 里，这种办法被称为 sprint 回顾会议。在会上，团队一起讨论在上个 sprint 大家什么做得好以及什么可以做得更好。在这上面花点时间来保证你每天的工作都是在做正确的事情。

一个常见的问题是，spring 回顾会议的结果并没有真正地被执行。很不幸，这可能是由于你和企业的其他部分沟通不畅的问题导致的。所以，这些问题在回顾会议上虐你千百遍，但是从来没有得到解决。

如果你意识到你的团队正处于这种情况，你将会从 DevOps 方法中获益，因为它强调企业内部的协作。

总结一下，尝试使用敏捷方法提供的机制。如果你正在使用 Scrum，请使用 sprint 回顾会议机制来捕获潜在的改进空间。话虽如此，这些方法不是教条，找出适合你的方法。

DevOps 和 ITIL（信息技术基础架构库）

这一部分说明了在一个大整体中，DevOps 和其他的工作方式怎么共存并互相适应。

DevOps 在敏捷或者精益企业的许多框架里都能协作得很好。大规模敏捷框架，或者说 SAFe®，都特意提到了 DevOps。自从 DevOps 在敏捷环境中诞生以来，各种各样的敏捷实践和 DevOps 之间就几乎从来没有过分歧。然而，ITIL 有些不同。

ITIL，信息技术基础架构库（Information Technology Infrastructure Library），是很多大型成熟企业采用的一种实践。

ITIL 是个扮演了软件生命周期中许多角色的大型框架。DevOps 和持续交付认为我们交付到生产环境的变更集应该小而频繁，乍一看，ITIL 的观点似乎与之相反。然而这并不是事实。遗留系统通常是单块系统，在这些系统中，经常伴随着复杂变更，需要有一个像 ITIL 一样的流程来管理它们。

如果你正在一个大型企业里工作，那么很可能你正在和这样的大单块遗留系统一起工作。

还有，许多 ITIL 描述的实践可以直接转换成相对应的 DevOps 实践。ITIL 规定了配置管理系统和配置管理数据库，这些类型的系统也是 DevOps 必不可少的一部分。本书将会介绍它们中的某些系统。

总结

本章概述了 DevOps 运动背景。探讨了 DevOps 的历史和它在开发和运维中的起源，以及敏捷运动。我们也了解了在大型企业中 ITIL 和 DevOps 如何共存，同时还讨论了怎样来避免草包族的反模式。你现在应该能够回答如何在大规模敏捷背景和不同的敏捷开发周期中使用 DevOps。

我们会逐渐地过渡到更倾向于技术和实践的主题。下一章我们将会介绍在 DevOps 中尝试关注的技术系统都是什么样子的。

2
洞察全局

DevOps 流程和持续交付流水线可能非常复杂。在开始实践之前，你需要搞清楚最终需要的结果是什么。

本章将会帮助你了解持续交付流水线的多个系统如何协同工作，形成对 DevOps 整体的概念。

在这一章，我们将会看到：

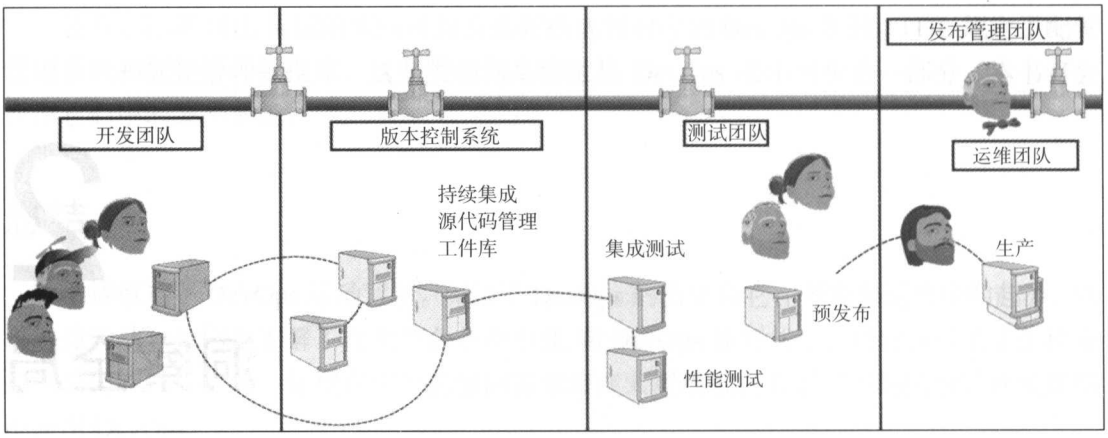
- DevOps 流程概览，一个持续交付流水线的实现，还有流程参与者。
- 发布管理。
- Scrum、看板和交付流水线。
- 瓶颈。

DevOps 流程和持续交付——概览

下面这张持续交付流水线概览图上的细节太多，以至于你可能不会细读所有的文字。现在先不用着急，我们将会一路深入了解。

就眼下来说，了解这一点就足够了：和 DevOps 一起工作时，意味着我们工作在一个又庞大又复杂的背景下，在一个又庞大又复杂的流程上。

下图描述了一个大型企业中，持续交付流水线的一个例子：



不论里面描述的是什么样的企业，这幅图的概览经常令人称奇。当然，对于不同的企业，还有开发的产品的复杂度，它们还是不一样的。

链条的前半部分，也就是开发环境和持续集成环境，一般都很相近。

测试环境的数量和类型会有很大的差异。生产环境也很不一样。

在下面的章节中，我们将会讨论持续交付流水线中的不同部分。

开发人员

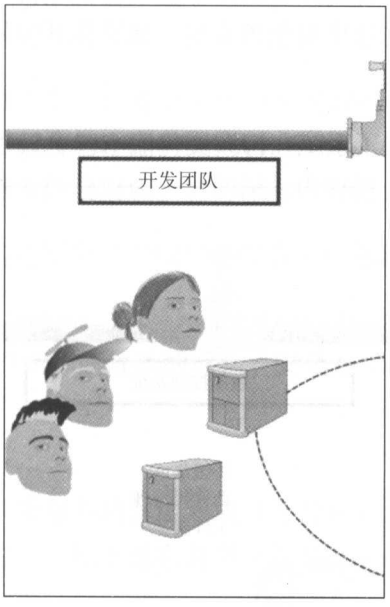
开发人员（上图的最左边）工作在自己的工作站上。他们开发代码并且需要许多工具来提高效率。

下图来自于上面那张持续交付流水线的大图，展示了开发团队。

理想情况下，他们中的每个人都应该在自己的台式机或笔记本电脑上拥有类生产环境。视开发的软件类型而定，也许可以做到，但更可能只是模拟甚至 mock 生产环境中很难复制的部分。例如，可能依赖于外部支付系统或者电话硬件。

和 DevOps 一起工作时，你在持续交付流水线上关注的部分可能取决于你原来所属的角色背景。如果你有很强的开发背景，你会喜欢像预打包开发环境带来的那种便利，并且愿意在这上面花费许多时间。这是一个合理的实践，否则开发人员可能需要花费大量时间来创建开发环境。这样的预打包环境可能包含着一个指定版本的 JDK（Java 开发工具包）

和 IDE（集成开发环境），比如 Eclipse。如果用到 Python，你也可以创建一个指定 Python 版本的包，诸如此类。



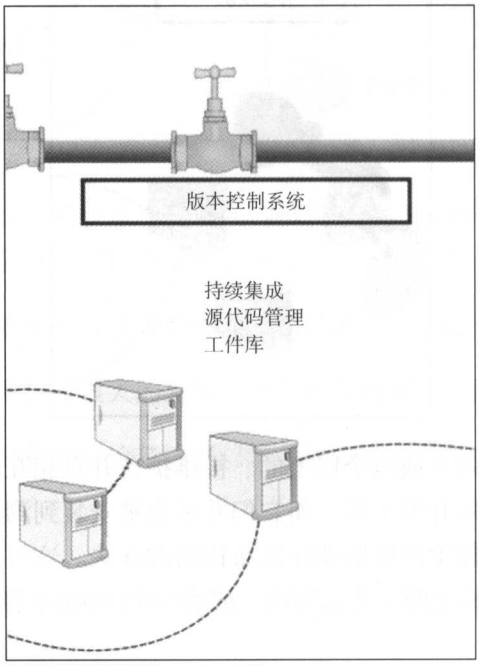
记住我们本质上需要两个或两个以上的、被维护着并且相互隔离的环境。先前的开发环境包含了所有我们需要的开发工具。而它们并不会被安装到测试或生产系统里。进一步说，开发人员也需要一些像生产环境那样发布代码的办法。这可以是开发人员的电脑上运行的 Vagrant 虚拟机、AWS 上的一个云实例，或者一个 Docker 容器——有很多办法都能解决这个问题。

我个人喜欢使用类似于生产环境那样的开发环境。如果生产环境运行 Red Hat Linux，开发环境上可以运行 CentOS Linux 或者 Fedora Linux。因为你可以在本地使用许多和生产环境一样的软件，麻烦少了，方便多了。使用 CentOS 或者 Fedora 的其他原因是 Red Had 软件许可证需要收费，并且企业级发行版上的软件通常有点儿滞后。如果你在生产环境使用 Windows 服务器，那么开发机上使用 Windows 也会很方便。

版本控制系统

版本控制系统一般来说是开发环境的中心。企业里软件产品的各部分代码都存放在这里。把配置和基础设施存放在这里也相当常见。如果你开发硬件，那么设计文档也可以存放在版本控制系统里。

下面的图片更详细地展示了系统在持续交付流水线上处理代码、持续集成和存储工件：



令人惊讶的是，企业的基础设施中这么关键的部分，可以选择的产品却很有限。近来，许多组织正在使用或者切换至 Git，尤其是正在使用中的专用系统快要到期了的企业。

不管你的企业里使用的是哪个版本控制系统，产品的选择只是大图上的一个方面。

你需要决定文件夹结构的约定和使用的分支策略。

如果有大量依赖的组件，每一个组件你都可以使用单独的库。

由于版本控制系统是开发链的关键，它的许多细节将会在第 5 章构建代码中介绍。

构建服务器

构建服务器在概念上非常简单。可以将它看成煮蛋计时器，定时或是用其他的机制触发，构建源代码。

最常用的模式是让构建服务器紧盯着版本控制系统的提交。当一个提交发生时，构建服务器就从版本控制系统上更新自己本地的源代码。随即，构建代码并运行测试来验证代码提交的质量。这个过程被称为持续集成。它的深入内容将会放在第 5 章构建代码里。

与代码库不同，现在的构建服务器领域里还没有明确地出现一名胜利者。

本书中，我们将会探讨 Jenkins，这是一个广泛使用的构建服务器的开源解决方案。Jenkins 能做到开箱即用，给你简单而健壮的体验。安装也相当容易。

工件库

当构建服务器确认了代码质量并将其编译成可交付物时，将这些编译好的二进制工件存放在一个库里是非常有用的。一般来说它有别于版本控制系统。

本质上，这些二进制代码库是通过 HTTP 协议操作的文件系统。一般来说，除了存储元数据，它们还提供了根据不同的类型和版本信息等来检索和索引的功能。

在 Java 世界里，一个流行的方案是 Sonatype Nexus。Nexus 并不只限于 Java 工件，例如 Jar 或者 Ear，而且还可以存放像 RPM 这样的操作系统工件、JavaScript 开发工件等。

亚马逊 S3 是一个可以用来存储二进制工件的键值数据库。一些构建系统，比如 Atlassian Bamboo，可以使用亚马逊 S3 来存储工件。S3 协议是开放的，也有可以部署在内部网络的开源实现。一个可选方案是 Ceph 分布式文件系统，它提供了兼容 S3 的对象存储。

接下来我们要谈到的包管理器，本质上也是一个工件库。

包管理器

开发中经常使用的各种 Linux 服务器在原理上类似，但是在实践上又有一些不同。

类 Red Hat 系统使用 RPM 格式的包。类 Debian 系统使用.deb 的格式。它们虽然功能类似，但是包格式不同。只用一条命令就可以从二进制库里下载并安装这些包到服务器上。

这样的命令被称为包管理器。

在 Red Hat 系统上，这条命令是 yum，或者是更新的 dnf。Debian 系的系统上是 aptitude/dpkg。

这些包管理系统最大的优势是能很容易地安装和升级，并且自动安装依赖。

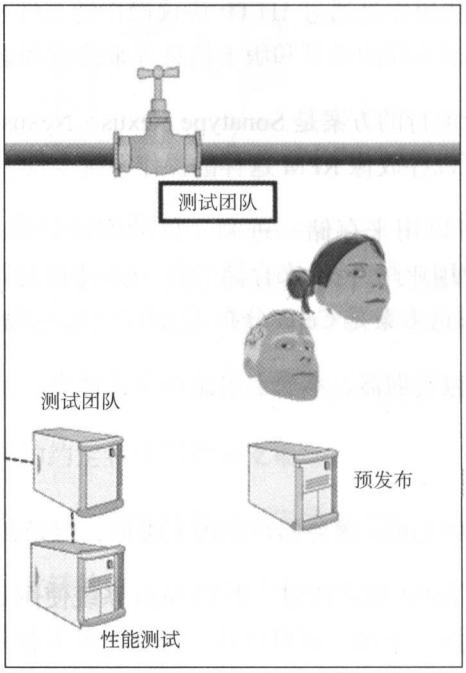
如果没有使用一个先进的系统，也可以远程登录进每一台服务器，然后输入 yum upgrade。最新的包就会从二进制库里下载安装。当然，我们将会看到，确实有更先进的可部署的系统。因此，我们再也不需要手动升级了。

测试环境

在构建服务器把工件存放在二进制库之后，它们就可以被安装到测试环境中。

下面的图更详细地展示了测试系统：

一般来说，测试环境应该尽可能像生产环境一样。所以，它们也应该能用相同方法在生产环境上安装和配置。



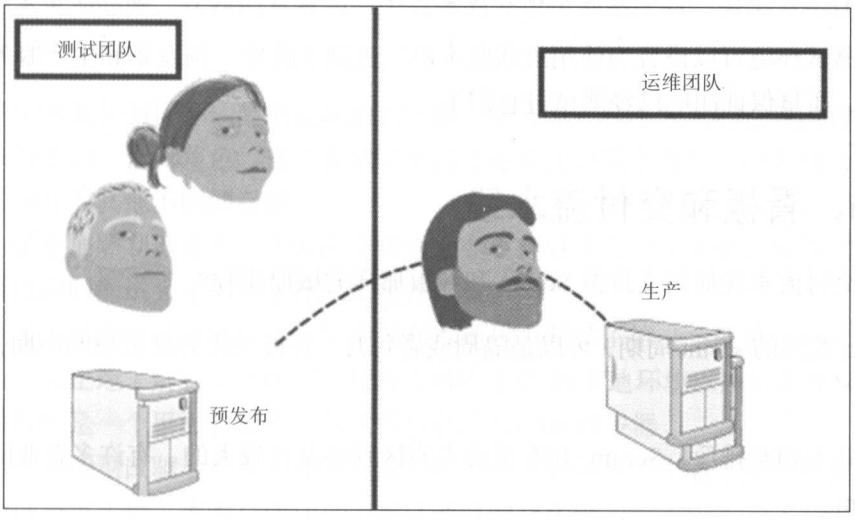
预发布/生产

预发布环境是测试环境的最后一关。它可以和生产环境互换。你把最新的发布安装到预发布服务器上，检查一切是否都正常，然后将老的生产环境切换过来，这样预发布环境就变成了新的生产环境。有时这被称为蓝绿发布策略。

这种部署方式的更详细做法因产品而异。有时，做不到让几个生产环境同时并行运行，因为生产环境通常都很贵。

另一方面，一个系统池里可能有着成百上千个生产系统。我们可以逐步在系统池里替换新版本。已登录的用户仍然停留在他们登录过的指定版本的服务器上。新用户将会登录到运行着软件新版本的服务器上。

持续交付图里的以下部分详述了最终的系统和参与的角色：



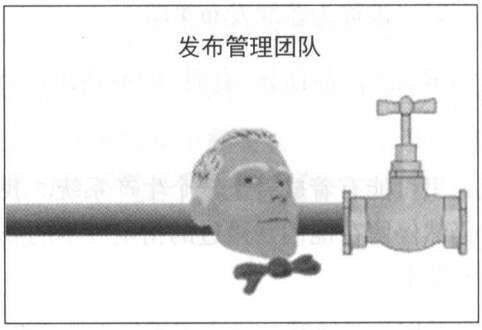
不是所有的企业都有资源来保持预发布环境和生产环境同步，但是只要有可能，用这样的方式来升级是优雅而可靠的。

发布管理

迄今为止我们一直假定发布过程主要是自动化的。这是 DevOps 梦寐以求的场景，而在真实世界中这个美梦是很难实现的。一个原因是需要相当高程度的自动化测试才能更有

信心实践自动化部署，而这通常很难做到。另一个原因是业务和技术开发的节奏不一定相同。所以，在发布过程中人工介入是必要的。

下图的水龙头代表专职发布经理的人工介入。



实践中的做法有很多，但是部署系统通常可以知道在不同的环境中使用的软件版本。

集成测试环境可以设置为使用最新版本的二进制工件库。预发布和生产服务器都有特定的版本，质量保证团队已经测试过它们了。

Scrum、看板和交付流水线

持续交付流水线如何支持像 Scrum 和看板那样的敏捷流程？

Scrum 关注的 sprint 周期，可以是隔周或者每月。看板关注的是更短的周期，可以说是每天。

虽然并不相互排斥，Scrum 和看板的本质区别还是比较大的。有许多企业同时使用看板和 Scrum。

从软件开发的视角来看，Scrum 和看板非常相似。它们都需要频繁地一站式发布。从 DevOps 的视角来看，一个变更从持续交付流水线开始，途经测试系统并最终通过，至此才认为它为整个旅程做好了准备。这样的判断可能是主观的或者是客观的，比如“所有的单元测试都绿了”。

一条流水线可以管理下面的两个场景：

- 构建服务器生成客观的代码质量指标，我们需要这些指标来做决策。这些决策可以自动生成，或者作为人工决策的依据。
- 部署流水线也可以由人工管理。可以由问题管理系统处理，或是通过提交配置代码，或是二者结合。

所以说，再一次从 DevOps 视角来看，不管我们用 Scrum、SAFe、看板，或是其他精益及敏捷框架都没什么太大关系。甚至传统的瀑布流程也能被成功管理——DevOps 适用于一切！

圆满结束——一个完整的例子

到目前为止，我们粗略地探讨了许多信息。

为了让它更加明确，让我们看看当一个具体的变更传到系统时，将会发生什么事。举个例子：

- 开发团队接到任务，要给企业的系统做一个变更。这个变更的主要内容是给鉴权系统增加一个新角色。这个看似简单的任务其实没那么容易，因为这个变更将会影响许多其他不同的系统。
- 为了更加顺利地开发，大家决定将这个变更拆分成几个小变更，这样它们就可以被以回归测试为主的自动化测试分开验证。
- 开发人员在自己的电脑上开发并且在本地尽力测试了第一个变更——新增角色。为了真正地了解它是否可用，开发人员需要他/她本地环境以外的系统权限。在这里指的是一个里面有用户和角色等信息的 LDAP 服务器。
- 如果使用测试驱动开发，在写实际代码之前会先编写一个通不过的测试。在这个测试写完之后，才编写能让这个测试通过的新代码。
- 开发人员将代码提交到企业内部的 Git 版本控制系统上。
- 构建服务器获取到了这个变更并初始化构建流程。单元测试之后，这个变更被认为可以被发布到 Nexus 的二进制库里。
- 配置管理系统 Puppet 发现鉴权组件有了一个新的版本。由于集成测试服务器被配置为总是使用最新的版本，于是 Puppet 勇往直前安装了最新的组件。
- 新组件的安装触发了自动化回归测试。在这些测试成功结束之后，质量保证团队就开始做人工测试。

- 质量保证团队给这个变更盖上“已通过”的章。变更转向预发布服务器，在这里开始了最后的验收测试。
- 当验收测试完成后，预发布服务器被切换成了生产环境，而生产环境转变成了新的预发布环境。企业的负载均衡服务器管理着最后的这一步。

这个流程按需一遍遍地重复着。就像你看到的那样，有一大堆的事情呢。

识别瓶颈

任何从开发到生产通过流水线的变更，都有许多事情，就像上一个例子那样。把这个流程变得更有效率是非常重要的。

和所有的敏捷工作一样，时刻关注你正在做什么，尝试识别问题范围。

如果一切工作正常，对代码库进行提交时，应该能够在 15 分钟内把变更部署到集成测试服务器上。

如果不那么正常，一次部署可能带来几天预期之外的烦恼。这里列举了一些可能的原因：

- 数据库结构变更。
- 测试数据与预期不匹配。
- 部署依赖于某人，而这个人没空。
- 变更伴随着一堆没有实际作用的官僚流程。
- 你的变更太大了，所以为了安全部署，需要做一大堆功课。这可能是由于你的架构是个单块系统。

后面的章节中，我们将会更深入地评审这些问题。

总结

在本章，我们更深入地探究了当你从事 DevOps 时，在不同类型的系统和流程中的日常工作。我们对 DevOps 的核心——持续交付流程有了更深入和详尽的理解。

接下来，为了更快、更准确地交付，我们将会研究 DevOps 理念是怎样影响到软件架构的。

3

DevOps 如何影响架构

软件架构是一个非常广的主题，在本书中我们将会关注持续交付和 DevOps 上明显相互作用的架构部分。

在本章，我们将会看到：

- 软件架构观点及其在我们以 DevOps 的视角工作时对我们的意义。
- 基本术语和目标。
- 反模式，比如单块系统。
- 关注点分离的基本原则。
- 三层系统和微服务。

我们最后会在诸如数据库迁移这样的实际问题上结束。这可是非常棘手的事，赶紧开始吧！

介绍软件架构

我们将会讨论 DevOps 怎样影响应用程序的架构，而不是书里其他部分讨论的软件部署系统的架构。

在讨论软件架构时，我们通常想到的是软件的非功能性需求方面。非功能性需求指的是软件的不同特性，而非指定行为的需求。

系统能够处理信用卡交易，这是一个功能性需求。系统每秒钟可以处理多少笔交易，这就是一个非功能性需求。

DevOps 和持续交付着眼于软件架构的两个非功能性需求：

- 我们需要频繁交付小的变更。
- 我们需要对质量有大的信心。

典型的场景应该是：我们可以持续交付许多的小变更，使其在较短的时间内从开发者的电脑上部署到生产环境。由于意料之外的问题导致回滚某个变更的可能性应该是微乎其微的。

所以，如果暂时把部署系统从这样的场景剔除，我们部署的软件系统架构会发生怎样的变化？

单块系统场景

理解问题架构给持续交付带来的难题，一种方式就是举个反例。

让我们假设有一个大的 web 应用程序，它有许多不同的功能。

在这个应用里有一个静态网站。整个 web 应用部署成一个单独的 Java 企业版应用程序。所以，如果只是想改正一个静态网站的拼写错误，我们就需要重新构建整个网络应用，然后重新部署。

虽然这个例子看上去很蠢，有见识的读者都不会这么干，但是我还真看到过这样的反模式。作为 DevOps 工程师，这可能是我们要解决的真实场景。

让我们把上面这团乱麻分解一下。在想要改正拼写错误的时候发生了什么？让我们来看一看：

1. 虽然知道拼写错误是哪一个，但是我们需要修改哪一个代码库呢？因为这是一个单块系统，我们需要在代码库的版本控制系统里创建一个分支。这个新分支与生产环境的代码相符。

2. 新建分支并改正拼写错误。

3. 用修改后的代码构建一个新的工件。赋给它一个新版本号。

4. 将这个新工件部署到生产环境。

好了，乍一看并不是太坏。但是考虑一下：

- 变更是在整个业务系统上做的。如果我们在部署新版本的时候出了什么错，其间的每分钟都会遭受损失。我们真的那么肯定这个变更不会影响其他部分？
- 事实上并不只是改正了拼写错误。我们还在生成新成品的时候改变了版本号。但是改变版本号应该是安全的，对吧？你确定吗？

这里的关键是我们已经在确认变更是否安全这件事上费了相当大的精力。系统太复杂了，即使考虑微不足道的变更所带来的影响也变得相当困难。

现实中，一个变更通常要比改正拼写错误要复杂得多。所以，我们需要为所有变更考虑部署链上包括人工校验在内的所有方面。

这样一来我们就到了一个不该去的地方。

架构经验法则

有一些架构法则可以帮助我们处理上文描述的恶劣情形。

关注点分离

著名的荷兰计算机科学家 Edsger Dijkstra 在 1974 年的论文论科学思维的作用（On the role of scientific thought）上，第一次提到了他关于怎样有效思考的观点。

他把这个观点称为“关注点分离”。迄今为止，它可以说是软件设计中最重要的原则。还有很多其他知名的法则，但是大多数都是从关注点分离原则中推导出来的。最基本的原则很简单：我们应该分开考虑系统不同的方面。

内聚原则

在计算机科学里，内聚指的是软件模块的各部分之间相互关联的程度。

内聚可以用来衡量模块内部的函数关联的强度。

模块内的高内聚是可取的。

我们可以看到高内聚是关注点分离原则的另一方面。

耦合

耦合指的是两个模块间相互依赖的程度。我们总是想要模块间低耦合。

我们可以再次看到耦合是关注点分离原则的另一方面。

高内聚低耦合的系统自带关注点分离，反之亦然。

回到单块系统场景

上一个改正拼写的场景里，很明显我们败在了关注点分离上。至少从部署的角度上看，我们完全没有任何的模块化。系统看上去都是低内聚高耦合的糟糕功能。

如果我们有一套分开的部署模块，拼写改正应该只会影响一个模块。很明显这样的部署变更更加安全。

当然实践中应该怎样来实现尚无定论。在这个特别的例子里，改正拼写很可能属于一个前端网络组件。最起码，这个前端组件应该能够从后端分离出来单独部署，并且拥有它们自己的生命周期。

然而在真实的世界里，我们可能没有足够幸运到总是能够影响企业采用不同的技术。例如，前端可能是由一个专门的内容管理系统用自创的招式实现的。如果你遇到这种情况，明智的做法是时刻关注这样的系统所带来的成本。

一个真实例子

现在让我们来看一个真实的例子，本书的剩余部分里将经常用到它。在这个例子里，我们为一个名为 Matangle 的企业工作。这个企业是一个软件即服务（SaaS）提供商，给学生售卖教育游戏。

就像所有的提供商那样，十有八九会有一个客户信息数据库。这个数据库就是我们的起点。

企业的其他系统将会随着我们的前进而呈现，这个初始系统目前还是挺合适的。

三层系统

Matangle 的客户数据库是很典型的 CRUD（**创建、读取、更新和删除**）类型的三层系统。在过去的数十年前就使用了这种软件架构风格，而且一直还很流行。这类系统非常普遍，你很有可能会碰上一个，不管是遗留系统还是全新系统。

在这张图中，我们可以看到关注点分离的实践：

下面列举的三层展示了一个企业如何构建这个系统的例子。

表示层

表示层将会是一个使用 React 框架的网络前端。它会被部署成一套 JavaScript 和静态 HTML 文件。React 框架相当新潮。你所在的企业可能不会使用 React，但也可能会用例如 Angular 的其他框架来替代。不管怎样，从部署和构建方面来看，大多数的 JavaScript 框架都很类似。

业务层

业务层是一个使用 Java 平台上的 Clojure 语言实现的后端。Java 平台在大型企业中非常流行，较小的企业可能倾向于使用其他基于 Ruby 或者 Python 的平台。我们的例子基于 Clojure，有点儿合二为一的感觉。

数据层

在我们的例子里，数据库是用 PostgreSQL 来实现的。PostgreSQL 是一个关系型数据库管理系统。它确实不像 MySQL 那么流行，大型企业可能喜欢用 Oracle 数据库。而 PostgreSQL，在任何情况下都是一个健壮性很强的系统，这就是企业选择它的原因。

从一个 DevOps 的观点来看，这个三层风格起码在表面上看起来像那么回事。应该可以为这三层中的每一层单独地部署变更，这会让小变比较容易地部署上服务器。

不过实践中，数据层和业务层经常紧紧地耦合在一起。对于表示层和业务层来说也一样。为了避免这样，必须小心地保持层之间的接口精简。使用知名的套路并不是万能药。如果设计系统时不够小心，我们仍然会以意料之外的单块系统而告终。

处理数据库迁移

处理关系型数据库的变更需要特别注意。

关系型数据库既存储数据又存储数据结构。升级数据库与升级程序相比有一些其他的挑战。一般来说，当升级一个程序的二进制文件时，我们停止应用程序，再升级，然后再启动它。我们并不太在意应用的状态，那是在程序之外处理的。

当升级数据库时，我们需要考虑状态。因为一个数据库几乎没有什么逻辑和结构，但是有许多的状态。

为了描述一个数据库的结构变更，我们发出一条命令来改变结构。

数据库结构变更前后，在数据库里需要能够查看到各自的版本。我们怎么来维护数据

库的版本呢?

Liquibase 是一个数据库迁移系统，其内核久经考验。像这样的系统有很多，一般来说至少每种编程语言都会有一种。Liquibase 在 Java 中很有名气，即便如此，在 Java 世界中也还有一些类似的其他选择。Flyway 是 Java 平台的另一个例子。

一般来说，数据库迁移系统会使用以下方法或变体：

- 往数据库里增加一张表，用于存放数据库的版本。
- 把数据库变更的命令统一存放到带版本信息的变更集中。在 Liquibase 里，这些变更被保存为 XML 文件。Flyway 实现方式有点儿不太一样，这些变更集被保存为 SQL 文件。万一过于复杂，也可以是单独的 Java 类。
- 当 Liquibase 需要升级一个数据库时，它会查看元数据表并决定这些变更集的顺序以将数据库升级成最新版本。

如前文所述，许多数据库管理系统都如此运转。它们最主要的不同一般是变更集的保存方式，以及如何决定运行哪一个变更集。它们可能会像 Liquibase 那样保存为 XML 文件，也可能像 Flyway 那样保存为 SQL 文件。原生系统更经常使用后者，它还有一些优点。Clojure 的生态系统也至少有一个类似的数据库迁移系统，称为 Migratus。

滚动升级

在迁移数据库时另一个需要考虑的事是如何配合滚动升级。这种类型的部署非常常见，尤其是在你不想让最终用户面临停机，或者只允许极低停机时间时。

这里有一个滚动升级企业客户数据库的例子。

一开始时，我们有一个运行中的系统，一个数据库和两台服务器。在两台服务器前面有一个负载均衡设备。

我们正准备推出一个数据库结构的变更，它会同时影响到服务器。我们准备把数据库中的客户名字段分成姓和名两个单独的字段。

这是一个不兼容的变更。怎样才能减少停机时间？让我们来看看解决方案：

1. 我们始于数据库迁移：生成两个新字段，然后将旧的名字用空格分成两部分，填充到这两个新字段中。旧的名字只是项目初始的选择，现在并不是非常适合，这是我们想要改变的原因。

   这个变更到现在为止还是向下兼容的，因为我们并没有移除名字字段，我们只是新建了两个还没有用到的新字段罢了。

2. 接下来，我们修改负载均衡的配置，从而使外界不能访问我们的第二个服务器。第一个服务器依旧当仁不让，因为旧的服务器代码仍然可以访问旧的名字字段。

3. 现在第二个服务器没有负载了，我们可以随意升级。

   升级完成以后，我们启动它。因为使用了两个新字段，它也工作得很好。

4. 此时，我们可以再次切换负载均衡的配置，使第一个服务器变得不可用，而由第二个服务器提供服务。当第一个服务器断线的时候，我们也可以照样升级它。再次启动它以后，通过恢复负载均衡的配置让两个服务器都可以被访问。

现在，这个变更就差不多完成了。剩下的唯一事情就是移除旧的名字字段，因为没有代码会再去使用它。

就像我们所看到的，为了正常工作，滚动升级需要预先的大量工作。如果你的企业本来就有停机计划，在停机期间升级要容易得多。国际企业可能并没有合适的时间窗口来处理升级，滚动升级可能是唯一的选择。

Liquibase 的 Hello world

这是一个基于 Liquibase 关系型数据库变更集的“hello world”简单例子。

想要试验的话，解压源代码后运行 Java 的构建工具 Maven。

```
cd ch3-liquibase-helloworld
mvn liquibase:update
```

变更记录文件

下面是一个 Liquibase 的变更记录文件的例子。

它定义了两个变更集，或者说是迁移，id 为 1 和 2：

- 变更集 1 创建了一张名为 customer 的表，有一个 name 字段。
- 变更集 2 对 customer 这张表增加了一个 address 列。

```
<?xml version="1.0" encoding="utf-8"?>
<databaseChangeLog
xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.0.xsd">

  <changeSet id="1" author="jave">

    <createTable tableName="customer">
        <column name="id" type="int"/>
        <column name="name" type="varchar(50)"/>
      </createTable>
  </changeSet>
  <changeSet id="2" author="jave">
        <addColumn  tableName="customer">
        <column name="address" type="varchar(255)"/>
        </addColumn>
  </changeSet>

  </databaseChangeLog>
```

pom.xml 文件

Maven 的标准项目文件是 pom.xml，定义了诸如需要连接的数据库的 JDBC URL、Liquibase 的插件版本之类的东西，以便我们能够在这个数据库上工作。

我们会创建一个名为/tmp/liquidhello.h2.db 的 H2 数据库文件。H2 是一个对

测试友好的内存数据库。

这个 pom.xml 文件就是“liquibase hello world”的例子：

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>se.verona.liquibasehello</groupId>
  <artifactId>liquibasehello</artifactId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.liquibase</groupId>
        <artifactId>liquibase-maven-plugin</artifactId>
        <version>3.0.0-rc1</version>
        <configuration>
          <changeLogFile>src/main/resources/db-changelog.xml
          </changeLogFile>
          <driver>org.h2.Driver</driver>
          <url>jdbc:h2:liquidhello</url>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>1.3.171</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

如果你运行这段代码并且一切工作正常，得到的结果就是一个 H2 数据库文件。H2 提

供一个简单的 web 界面，你可以在那里验证数据库的结构是不是你所期望的。

手动安装

在做自动化之前，我们需要了解相对应的手动流程。

本书假设我们正在使用 Red Hat Linux 发行版，例如 Fedora 或者 CentOS。许多 Linux 发行版的本质上都是类似的，除了包操作的一些命令可能会略有不同。

就练习而言，你可以使用物理机或者 VirtualBox 的虚拟机。

首先我们需要安装 PostgreSQL 关系型数据库。使用以下命令：

```
dnf install postgresql
```

它会检查是否已经安装了 PostgreSQL 服务器。否则，它会从远程的 yum 库里下载并安装 PostgreSQL 包。所以仔细想想，其实许多潜在的手动步骤已经被自动化过了。我们不需要编译软件、检查版本、安装依赖等。所有的这些都已经在 Fedora 项目的构建服务器上预先完成了，非常方便。

不过为了自己企业的软件，最终我们也需要来学着做这些事情。

同样我们也需要一个网络服务器，例如在这种情况下的 NGINX。使用以下命令来安装：

```
dnf install nginx
```

在 Red Hat 发行版中，`dnf` 命令代替了 `yum`。它重写并兼容 `yum`，所以可以继续使用相同接口的命令。

我们正在部署的 Matangle 客户关系数据库，严格来说并不需要一个单独的数据库和网络服务器。在这个软件的 Clojure 层已经包含了一个称为 HTTP Kit 的网络服务器。

通常，在 Java、Python 还有其他服务器之前会用一个专门的网络服务器，最主要的原因是关注点分离。这一次，并不是因为业务隔离，而是非功能性需求，比如性能、负载均衡和安全上的考虑。目前，基于 Java 的网络服务器可能可以完美地提供静态内容，但是一个基于纯 C 语言的网络服务器，例如 Apache `httpd` 或者 NGINX 的性能更出众，内存更节省。使用一个前端网络服务器也很常见，例如 SSL 加速和负载均衡。

现在我们有一个数据库和一个网络服务器了。接下来需要构建和部署企业的应用程序。

在你的开发机上，在本书解压后的源代码文件夹里运行以下命令：

```
cd ch3/crm1
lein build
```

我们现在创建了一个 Java 程序，可以部署和运行了。

尝试启动应用程序：

```
lein run
```

在一个浏览器里访问终端输出的 URL 就可以看到 web 界面。

怎样才能正确地把应用程序部署到服务器上呢？如果我们可以使用与刚才安装数据库和网络服务器相同的命令和结构，那当然是最好了。我们将在第 7 章部署代码里介绍方法。现在的话，在 shell 里运行这个应用已经足矣。

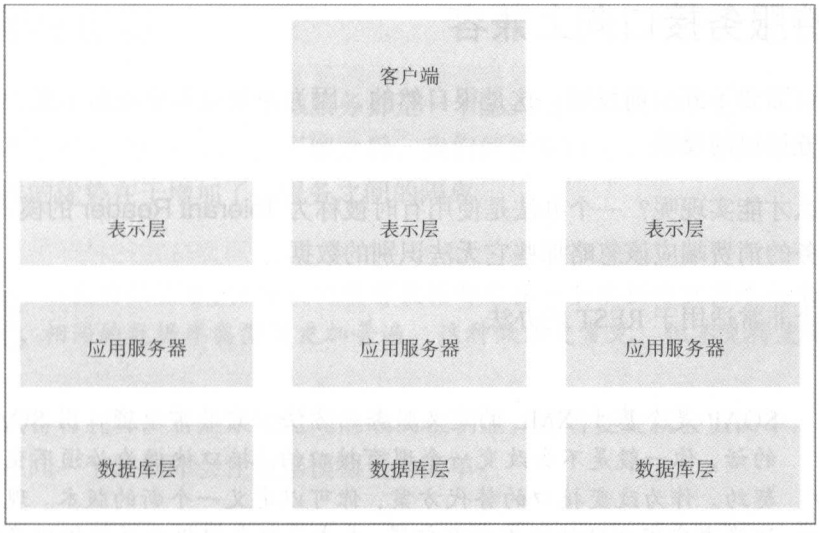
微服务

微服务是一个新兴的术语，用来描述这样的系统：三层架构的业务层由许多小的服务组成，它们之间使用语言无关的协议来通信。

一般来说，这种语言无关的协议是基于 HTTP 的，通常是 JSON REST，但是并不强制。协议层还是有选择余地的。

这种架构设计非常适用于持续交付方案，因为就像我们看到的那样，部署一些小而独立的服务比部署一个单块系统来说要更加容易。

下面这张图描述了一个微服务的部署看起来是什么样子的：



接下来，我们还会继续深入到微服务架构的例子里。

小插曲——康威定律

1968 年，Melvin Conway 提出一个观点，那就是设计软件的组织结构，等价于软件的组织架构。这被称为康威定律。

比如说三层架构，就反映出许多企业的 IT 部门结构：

- 数据库管理员团队，简称 DBA 团队。
- 后台开发团队。
- 前台开发团队。
- 运维团队。

噢，变成四个团队啦。但是我们可以很清晰地看到架构和企业的相似性。

DevOps 的主要目标是与不同的角色共同协作，最好是一个跨职能团队。如果康威定律是正确的，这种团队的企业将会反映到他们的设计里。

微服务模式正好密切反映了跨职能团队。

如何保持服务接口向上兼容

服务接口需要不断向前发展。这是很自然的，因为企业也需要向前发展，而在相当程度上它是服务接口的反映。

我们怎么才能实现呢？一个办法是使用有时被称为 Tolerant Reader 的模式。它的含义很简单：服务的消费端应该忽略那些它无法识别的数据。

这是一个非常适用于 REST 的办法。

SOAP 是个基于 XML 的定义服务的方法，它非常缜密。用 SOAP 的话，你一般是不会改变一个现有接口的。接口被视为必须不变的契约。作为改变接口的替代方案，你可以定义一个新的版本。现有消费者实现新的协议再重新部署，或者是服务提供方并行提供多种版本的接口。这种方法比较烦琐并且在服务提供端和消费端造成了讨厌的紧耦合。

DevOps 和持续交付并不强制应该怎么做事情，所以最有效率的方法更令人中意。

在我们的例子里，可以说把变更的实现分摊到生产者和消费者里是代价最小的办法。不管怎样，生产者需要变化，而消费者需要接受实现 Tolerant Reader 的一次性开销。通过 SOAP 和 XML 来实现也是可以的，但是相比 REST 来说显得不那么自然。这也是为什么在拥抱 DevOps 和持续交付的企业里，REST 的实现更加流行的一个原因。

如何实现 Tolerant Reader 模式在实践中因平台而异。对 JsonRest 来说，通常把 JSON 结构转换成同等的语言结构就足够了。你的程序需要哪些部分，你就用哪些部分。所有的其他部分，不管是旧的还是新的，通通忽略掉。这种办法的局限是，生产者不能移除消费者依赖的部分。增加新部分没有问题，因为它们将会被忽略。

这样又给生产者增加了负担，它必须记住哪些是消费者需要的。

在企业的高墙内，这并不是什么大问题。生产者可以知道消费者最新的代码并在构建生产者的阶段进行测试。

而对于那些暴露在因特网上的服务，这种方法并不那么实用，这时会倾向于使用更为严格的 SOAP 方式。

微服务和数据层

一种看待微服务的方式是每个微服务都是一个隐式的三层独立系统。不过我们通常不为每一个微服务都实现所有的层。了解之后，我们便能发现每个微服务都可以实现自己的数据层。这样的优势在于增加了各服务之间的隔离。

以我的经验看，把企业的所有数据都放在一个单独的数据库或至少相同的数据库类型里更加普遍。这种做法更常见，但不见得更好。

两种方式各有利弊。若是系统之间的隔离很明显，部署变更就会更简单。反之，把所有数据都存在同一个数据库会让数据模型更为简单。

DevOps、架构和弹性

我们已经从 DevOps 的角度看到微服务架构有许多值得拥有的特质。DevOps 的一个重要目标是更快地为用户交付新特性。这是微服务提供的大量模块化所带来的结果。

那些担心微服务会提供一个毫无瑕疵的完美解决方案从而让生活变得没意思的人可以解脱了。微服务有它自己的挑战。

我们想要能够尽快部署新代码，但是我们也想要可靠的软件。

微服务在系统间有更多的集成点，比起单块系统来说更有可能失败。

DevOps 的自动化测试非常重要，这样我们部署的变更才能有更好的质量，才能令我们更加信赖。然而，这并不是一个可以解决服务由于不明原因突然宕机的方案。由于在微服务模式中我们有更多的服务，从统计学上来说服务宕机的概率更高。

我们可以通过努力监控服务并在出状况时采取适当的行动来部分缓解这个问题。最好是自动化的方式。

在我们的客户数据库例子里，可以采用以下策略：

- 使用两个应用服务器同时运行应用程序。
- 应用程序通过 JsonRest 提供特定的监控接口。

- 监控后台定时调用监控接口。
- 如果服务器停止工作，负载均衡便会重新配置以将其从服务器池中移除。

显然这是一个简单的例子，但它有助于描述我们面临的挑战，那就是设计由许多动态部分组成的弹性系统以及它们如何影响架构决策。

为什么要针对不同的应用程序来提供监控接口？这是因为监控的目标是让我们全面了解当前系统的健康状态。我们一般监控应用程序栈的许多方面。监控服务器的 CPU 没有过载，还有足够的磁盘和内存空间可用，基础的应用服务器还在运行，等等。尽管通过这些，可能还不足以断定服务运行正常。例如，可能由于某些原因，服务的数据库配置错误。一个针对服务的健康检查接口可以尝试连接数据库并在结果中返回连接的状态。

当然，如果你所在的企业同意以通用结构来检查返回值，那就再好不过了。这个结构也会因使用的监控软件类型而异。

总结

在本章，我们用 DevOps 和持续交付的视角了解了大量关于软件架构的话题。

我们学习了关注点分离原则带来的许多不同方面。我们也开始在企业内部组件 Matangle 客户数据库的部署策略上工作。

我们深入到细节里，例如如何从库里安装软件以及如何管理数据库变更。我们还见识了高层次的主题诸如经典的三层系统和更加时髦的微服务概念。

下一章我们将会讲述如何管理源代码和配置源代码版本控制系统。

4

一切皆代码

一切皆代码，而你需要一个地方来存储。这个地方就是企业里的源代码管理系统。

对不同类型的代码，开发和运维人员共享同一个集中式存储。

有许多方法可以提供集中式代码库：

- 可以使用软件即服务的解决方案，例如 GitHub、Bitbucket 或者 GitLab。性价比高，可用性好。
- 可以使用云提供商，例如 AWS 或者 Rackspace，来提供代码库。

有些企业不允许它们的代码离境。对于它们来说，一个私有的内部系统是最好的选择。

本章我们将会探讨不同的方案，例如 Git，还有基于 web 的 Git 前端，例如 Gerrit 和 GitLab。

本章我们将会开始经历 DevOps 领域的一个挑战：可供选择和研究方案太多！在 DevOps 的中心——源代码管理领域里尤其如此。

因此，我们将会从用户的角度，引入软件虚拟化工具 Docker，以便在探索中使用。

源代码控制的必要性

Terence McKenna，一位美国作家，曾经说过“一切皆代码”。

可能有人会不同意 McKenna 关于宇宙中的一切都可以用代码表示的说法，不过就

DevOps 而言，确实几乎一切都可以用代码的形式来表达，包括以下部分：

- 我们构建的应用程序。
- 部署应用程序的基础设施。
- 产品文档。

甚至运行应用的硬件也可以用软件的形式来表达。

鉴于代码的重要性，我们放置代码的地方，也就是代码库，理所应当是企业的中心。我们制造的几乎一切都在其生命周期的某个阶段途经代码库。

源代码管理历史

为了了解源代码控制的重要性，对源代码管理的开发历史来一个简短回顾应该会有一些启发。这样能让我们洞察到底需要什么特性。以下是一些例子：

- 把源代码的历史版本存放在不同的档案中。

  这种方式最简单，某种程度上还存在着，许多免费软件项目提供过去发布的 tar 档案供下载。

- 签入和签出的集中式源代码管理。

  在一些系统里，一个开发者可以锁住文件独家专享。每一个文件都被单独管理。像这样的工具包括修订控制系统（RCS）和源代码控制系统（SCCS）。

一般来说，你不再会碰上这类工具。除了偶尔的文件头表明了这个文件曾经被 RCS 管理过。

- 一个集中式的存储，在提交之前需要合并。包括**并发版本系统（CVS）**和 **Subversion**。

  尤其是 Subversion，现在仍然用得很广。许多企业有集中式的工作流程，而 Subversion 能帮它们很好地实现这样的流程。

- 一个去中心化的存储。

> 在进化阶梯的每一步，我们都得到了更多的灵活性、并发、更快的速度和更有效率的流程。我们也得到了可以自毁长城的更高级和强大的武器，切记！

目前，Git 是这个类别里最流行的工具，但是还有许多其他类似的工具也还被使用着，诸如 Bazaar 和 Mercurial。

时间会告诉我们 Git 和它的数据模型是否会阻挡住其他源代码管理宝座的竞争者，接下来的几年内，谁将会毫无疑问地证明自己。

角色和代码

从一个 DevOps 的视角来看，用好源代码管理工具非常重要。在某种意义上，许多不同的角色都会使用源代码管理工具。对于技术型角色来说更是如此，但是对于其他角色，例如项目管理人员，就不那么明显了。

开发者靠源代码管理来生活和呼吸。这是他们的饭碗。

接下来的几章我们也会看到，运维人员也喜欢通过代码、脚本或者其他产品的方式管理描述基础设施。基础设施的描述包括网络拓扑结构、需要在特定服务器上安装的软件版本，等等。

质量保证人员可以把他们编写的自动化测试存放在源代码库里。对于像诸如 Selenium 和 Junit，还有其他的许多软件测试框架来说，确实是这样的。

尽管如此，运行各种任务时需要的手动步骤还是一个关于文档的问题。与其说这是一个技术问题，倒不如说是一个心理学或文化问题。

虽然许多企业使用 wiki 方案，比如说像 wiki 引擎驱动的维基百科，但是仍然有大量文档还是以 Word 格式存放在共享文件夹和电子邮箱里。

这对于某些角色来说，找到可用文档确实很难，但对于另一些角色来说反倒是很容易。从 DevOps 视角来看，这实在是令人遗憾，企业应该要花点精力让所有角色都可以轻松访问文档。

选对 wiki 引擎的话，把所有文档都以 wiki 的形式存放在集中式源代码库里是可行的。

哪一个源代码管理系统？

已经有许多的源代码管理（SCM）系统了，但由于 SCM 在开发中的重要性，这类型的系统还会持续不断地被开发出来。

尽管如此，现在有一个最主要的系统，Git。

Git 有一个很有意思的故事：它是 Linus Torvalds 把 Linux 内核的开发从 BitKeeper 这个当时的专门系统移出时创建的。BitKeeper 的软件许可证发生了变化，使得用它来管理内核变得不切实际。

所以 Git 能够支持工作流程相当复杂的 Linux 内核开发，在基础技术的层面上足以胜任大多数企业的要求。

Git 相对于其他更早的系统来说最主要的优势是：它是一个分布式版本管理系统（DVCS）。还有许多其他的 DVCS，但是 Git 是使用最广的。

分布式版本管理系统有一些优点，包括但不限于以下几点：

- 就算没有联网，也可以高效地使用一个 DVCS。当你在火车上或者洲际航空时，可以带着一起工作。
- 因为不需要每个操作都连接到服务器上，DVCS 与其他 VCS 相比，在大多数场景里都运行得更快。
- 可以自己独立工作，直到你感觉可以把成果共享出去。
- 可以同时和多个远程环境协同工作，避免了单点故障。

除了 Git 以外，其他的分布式版本管理系统还有：

- **Bazaar**：缩写是 bzr。Bazaar 是由 Ubuntu 背后的 Canonical 公司认可和支持的。Canonical 的代码托管服务 Launchpad 支持 Bazaar。
- **Mercurial**：著名的开源项目比如 Firefox 和 OpenJDK 使用 Mercurial。它和 Git 差不多是同时期被开发的。

Git 可能很复杂，但是它的快速和高效弥补了这一缺点。它也可能很难懂，但是支持不同任务的前端可以让它变得更容易使用。

源代码管理系统迁移之言

我曾经用过许多源代码管理系统，并经历过许多次从一种系统迁移到其他系统。

有些时候，迁移的很多时间花在了保持历史记录完整性上。对于一些系统来说，这样的时间开销花得值，例如令人尊敬的免费或开源项目。

对于许多企业来说，保持历史记录并不值得花费那么多的时间与精力。如果有时需要旧版本，可以保留旧的源代码管理系统在线，以备参考。这包含了来自 Visual SourceSafe 和 ClearCase 的迁移。

有些迁移不值一提，例如从 Subversion 迁移到 Git。不会牺牲历史记录。

选择分支策略

如果工作的代码将会部署到服务器上，在企业间约定分支策略是很重要的事。

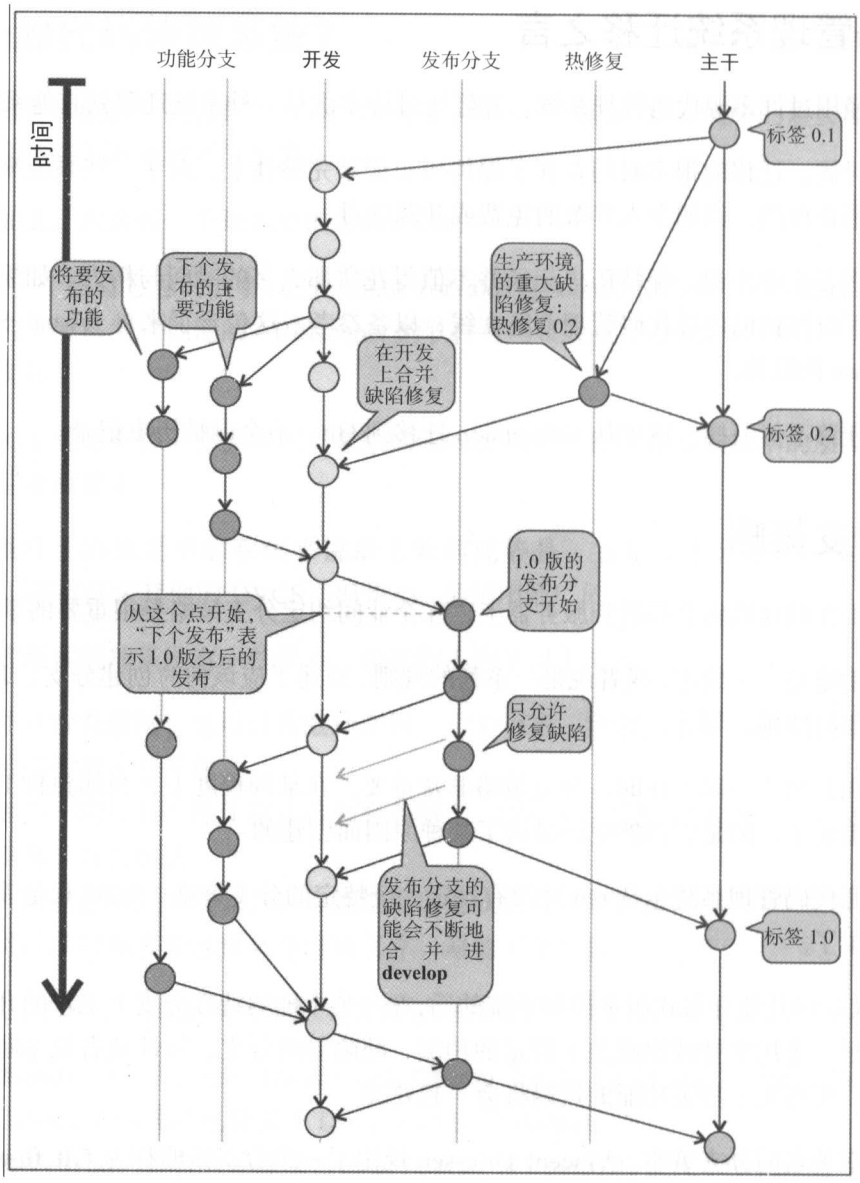
分支策略是一个约定，或者说是一系列的规则，描述了应该何时创建分支、如何命名、分支应该如何使用，等等。

当你和其他人一起工作时，分支策略非常重要。在某种程度上，当你独自工作时，它就没那么重要了，但是它仍然应该是为了某种原因而创建的。

许多源代码管理系统包括 Git 并没有规定一个特定的分支策略。SCM 只是简单地给你分支工具罢了。

使用 Git 和其他分布式版本控制系统的话，在一个本地的特性分支上工作的开销很小。分支仅仅是一个用来持续追踪关于特定的功能、缺陷等的分支、特性或者说主题而已。用这种方式，所有关于特定功能的代码将会一起处理。

有许多著名的分支策略。Vincent Driessen 规定了一个分支策略称为 **Git flow**，它有许多不错的特性。对于某些人来说，Git flow 太复杂了，在那些案例里，它是可以缩减的。还有许多可用的缩减版模型。Git flow 看起来的像下图：



Git flow 看上去很复杂，所以让我们简要地看一看分支是用来做什么的：

- **主干**分支只包含完成的工作。因为它们相当于发行版，所有的提交都打了标签。所有的发行版都源于主干。
- **开发**分支用来开发下一个发行版。当工作完成时，**开发**分支会被合并到主干。

- 我们为所有的新功能使用不同的**功能分支**。**功能分支**会被合并到开发分支。
- 如果生产环境出现了一个重大缺陷，会在开始修复缺陷时创建一个**热修复**分支。这个**热修复**分支之后会被合并到**主干**，然后被用来生成一个新的发行版。

Git flow 是一个集中式的模式，同样，它是 Subversion、CVS 流程的怀旧版。最主要的区别是使用 Git 会有一些技术和效率上的优势。

另一个策略，称为 forking 模式，就是每一个开发者都有一个集中式代码库，除了在企业内雇佣例如外包等外部团体以外很少使用。

分支问题域

在持续交付实践和分支策略之间有一个争论的根源。一些持续交付方法提倡一个主干分支，所有的发行版都从这个分支里生成。Git flow 就是这样的模型。

这么做简化了部署的一些方面，主要是因为分支图变得简单了。这反过来会让测试变得简单，因为只有一个分支会部署到生产服务器上。

另一方面，如果我们需要在已发布的代码里修复缺陷，而主干上还包含了我们不想发布的新功能时，应该怎么做呢？当生产环境的部署节奏比开发团队的交付节奏慢的时候，就会发生这样的场景。这种事令人不快，但还挺普遍。

有两种基本的处理方法：

- **创建一个缺陷修复分支并在其上部署到生产环境**：感觉上这样做比较简单一些，因为我们不会打断开发的流程。另一方面，这种方法可能需要双份测试资源。它们可能需要复制分支策略。
- **功能开关**：功能开关是另一种对开发者要求更严格的方法。在这种工作流程中，你关闭那些还未准备好上生产环境的功能。用这种方法可以发布最新的开发版，包括缺陷修复和暂时关闭的新功能。

选择哪种处理方法没有硬性规定，没办法教条地做决定。

最好同时为这两种场景做好准备，在特定环境里使用最适合于你的方法。

这里有些观点可以帮助你做出选择：
当你的变更主要都是向下兼容或者是全新的时候，功能开关是个好选择。否则，功能开关的代码可能会变得非常复杂，因为需要考虑许多不同的分支。反过来，还会让测试更加复杂。
缺陷修复分支使部署和测试变得复杂。虽然直观上可以创建分支来修复缺陷，但是新版本应该叫作什么，我们在哪里测试？测试资源很可能已经被正在进行中的开发分支占用了。

一些项目足够简单，我们可以凭空地创建测试环境。但是这样的项目很少，复杂的应用程序更多。有时测试资源非常稀缺，比如第三方的网络服务甚至是硬件资源，都很难被复制。

工件版本命名

如果你的程序比较大，版本号就变得非常重要。下面的列表展示了版本命名的基本原则：

- 版本号应该单向递增，也就是说，变大。
- 它们应该可以相互比较，这样就能轻易看出那个版本更新。
- 为所有工件使用相同的版本号体系。
  通常转换成带有三四个部分的版本号：
  - 第一部分代表了主要的代码变更。
  - 第二部分是次要变更，API 向后兼容。
  - 第三部分表示修复了缺陷。
  - 第四部分可以是一个构建号。

看起来挺简单，为语义化版本号（SemVer）生成标准可是相当费力的。完整的规格可以参考 `http://semver.org`。

如果所有的可安装工件都有一个合适的发布号和一个在源代码管理系统中相对应的标签，那是很方便的。

一些工具不是这么工作的。Maven，Java 的构建工具，支持快照发布号。一个快照可以被视为一个最后部分很含糊的版本号。快照有自己的一些缺点，比如说，你不能立即找到这个产品的源代码标签。这让调试宕掉了的程序变得更困难。

产品快照策略还违背了一个基本的测试原则：部署到生产环境的二进制产品应该与测试的产品完全一致。

当以快照方式工作时，直到测试完成之前你使用的一直是快照，之后又发布带有真实版本号的工件，然后那个工件才被部署上去。而它已经不再是相同的工件了。

当然，快照的基本前提是变更一个版本号不会带来实际影响，而墨菲定律表明了一定会有影响。以我的经验来看，墨菲定律在这里是正确的，快照版本瑜不掩瑕。还是使用构建版本号来代替吧。

选择一个客户端

Git 好的一方面是它并不强制使用特定的客户端，有好几个相互兼容的选项。大部分的客户端使用的是 Git 的核心实现方案之一，稳定性和质量都很好。

目前大部分的开发环境都能很好地支持使用 Git。

从这个意义上讲，选择一个客户端并不是我们必须要做的。大部分的客户端都工作得足够好，我们可以选择喜欢的客户端。许多开发者用的是集成在开发环境里的客户端，或者 Git 的命令行客户端。当执行运维任务时，大家更倾向于使用 Git 命令行客户端，因为在远程 SSH 的时候很方便。

一个需要我们做决定的例外情况是：当我们帮助企业里那些刚接触到源代码管理尤其是 Git 的人时。

在这些情况下，维护一篇使用说明很有帮助。里面记载了如何安装和使用一个简单的 Git 客户端，还包括配置连接到企业的服务器。

在企业的 wiki 上放一篇简单的使用说明手册一般能很好地解决这个问题。

创建一个基本的 Git 服务器

创建一个基本的 Git 服务器非常简单。虽然对一个大型企业来说它还不太够，不过在深入到更高级的方案之前这是很不错的练习。

首先让我们说明一下大致的步骤和需要完成的零碎：

1. 有两个账户的客户端机器。需要安装 `git` 和 `ssh` 包。

   SSH 协议是其他传输协议的重要基础，对 Git 来说也是一样。

   你手上需要有 SSH 公钥。如果由于某些原因没有，可以用 `ssh-keygen` 生成。

   我们需要两个用户，因为我们会模拟两个用户与一个中央服务器交互。为这两个测试用户生成秘钥。

2. 一台运行着 SSH 后台服务器。

   可以是模拟两个不同客户端用户的同一台机器，也可以是另一台。

3. 一个 Git 服务器用户。

   我们需要一个单独的 Git 用户来负责 Git 服务器功能。

   现在，你需要把两个用户的公钥添加到 `authorized_keys` 文件中，它在相对应用户的 `.ssh` 文件夹里。复制这个账户的密钥。

   是不是开始觉得这些事很麻烦？这就是一会我们要寻找一个简化版流程的原因。

4. 一个空的 Git 库。

   空的 Git 库是 Git 的一个特性。它们是 Git 库，只不过没有任何内容，所以只占用一小点儿空间。下面是如何新建：

```
sh cd /opt/git
mkdir project.git
```

```
cd project.git
git init --bare
```

5. 现在，尝试克隆，修改和推送上服务器。

让我们回顾一下方案：

- 方案的扩展性不是很好。

  如果只有两三个人需要操作，新建项目和增加密钥的工作还不是太经常，只是一次性开销罢了。如果你所在的企业里有许多人都需要如此操作，这个方案所需要的工作量就太大了。

- 解决安全问题会更麻烦。

  我觉得企业里限制员工访问系统的工作量太大了，虽然这个观点可能会有争议，但是不可否认在创建过程中你需要这个能力。

在这个方案里，你需要为不同的角色创建不同的 Git 服务器账号，而这将会是大量的重复性劳动。Git 并没有开箱即用的细粒度用户访问控制。

共享认证

大多数企业都有某个处理认证的中央服务器。LDAP 服务器是一个相当普遍的选择。假设你的企业已经想办法处理了这个核心问题并且运行了一个 LDAP 服务器，创建一个测试用的 LDAP 服务器还是相对比较容易的。

另一种它可能是使用 389 服务器，以 LDAP 服务器通常运行的端口命名，同时还提供一个 phpLDAPadmin 的 web 应用程序做认证。

对本书而言创建一个测试用 LDAP 服务器很有帮助，这样我们就可以为将要调研的不同服务器使用相同的 LDAP 服务器。

托管 Git 服务器

许多企业根本不允许使用其他企业托管的服务。

它们可能是政府机构或者与金钱打交道的企业，比如说银行、保险和游戏公司。

可能有法律方面的原因，或者也可以说只是一种对关键代码离开家门的紧张感。

如果你并不会感到不安，那么使用一个托管服务，比如 GitHub 或 GitLab 的私有账户，是很明智的。

无论如何，使用 GitHub 或 GitLab 是一个学习探索 Git 的捷径。

这两个供应商都非常容易评估，由于提供了免费账户，你可以借此了解它们所提供的服务。看看你是否真的需要所有的服务或者你可以凑合用点儿简单的。

GitLab 和 GitHub 比纯 Git 强的功能如下：

- web 界面。
- 内置 wiki 提供文档存放。
- 问题跟踪器。
- 提交可视化。
- 分支可视化。
- 拉请求工作流。

这些都是很有用的功能，但是你并不总会用到。例如，你可能已经有了一个 wiki、一个文档系统、一个问题跟踪软件等需要集成的东西。

我们正在寻找的，就是与管理和可视化代码息息相关的那些重要功能。

大的二进制文件

GitHub 和 GitLab 很相似，但是又有一些不同。其中之一源自诸如 Git 这样的源代码系统传统上并不太在意大的二进制文件的存储。总是有其他例如把文件服务器上的文件路径存放在纯文本文件里那样的办法。

在某种意义上，如果真的有二进制的源代码文件需要版本管理，该怎么办？这样的文件类型包括图像文件、视频文件、音频文件等。现代化的网站大量使用了媒体文件，而这个领域一贯是内容管理系统（CMSes）的地盘。CMSes 不管有多么好用，与 DevOps 相比还是处于下风的，所以把媒体文件存储在正常的源代码处理系统里具有很强的诱惑力。

CMSes 的缺点包括它们的脚本功能经常表现奇特或是不存在的事实。所以，另一个在 DevOps 工具箱里的词语——自动化，很难同 CMS 一起工作。

当然你可以直接把二进制文件提交给 Git，而它将会像其他文件那样被处理。接下来发生的事就是涉及服务器的 Git 操作突然就慢起来了。于是，Git 的主要优势——效率和速度——就这样被抛出窗外。

这个问题的解决方案随时间而发展，但是目前还没有明确的胜者。竞争者如下：

- **Git LFS**，GitHub 支持。
- **Git Annex**，GitLab 支持，但只是企业版。

Git Annex 更加成熟。这两个方案都开源并通过 Git 的插件机制实现。

还存在着一些其他的系统，这表明它是当前 Git 的一个尚未解决的痛点。Git Annex 在 `http://git-annex.branchable.com/not/`上比较了不同的方案。

如果需要对媒体文件进行版本控制，你应该开始考察 Git Annex。它用 Haskell 编写并且包含在许多发行版的包管理系统里。
还需注意的是这种方案的主要优势是对媒体文件和相对应的代码一起进行版本控制的能力。当与代码一起时，你可以很方便地检查不同版本间的代码。检查媒体文件的不同更难并且用处不太大。
简而言之，Git Annex 在数据逻辑问题上使用了成熟的方案；增加一个间接层。通过在代码库里存储文件的符号链接来实现。二进制文件被存放在文件系统里并被本地工作区以如 rsync 的其他方式所用。创建这个解决方案当然需要更多的工作量。

尝试不同的 Git 服务器实现

分布式的本质让为各种目的而尝试不同的 Git 实现成为可能。客户端的设定总是相似的，与服务器是怎么创建的无关。

你也可以并行使用几种方案。客户端不会因此而过分复杂，因为 Git 被设计为可处理不同的服务端。

中场休息，插播 Docker

在第 7 章部署代码中，我们将会看到用 Docker 这种令人激动的全新方式来打包应用程序。

本章要解决一个类似的难题。我们需要尝试几种不同的 Git 服务器实现，来看看哪一种最适合于我们的企业。

这可以通过 Docker 来实现，所以我们将会用这个机会来窥探 Docker 提供给的简单部署能力。

因为不久后会更加深入 Docker，本章我们会稍微糊弄点，宣称 Docker 是用来下载和运行软件的。这并不完全正确，但是 Docker 的功能比这样的描述要多得多。参照以下步骤开始使用 Docker：

1. 首先，按照操作系统的特定说明来安装 Docker。Red Hat 系列很简单，就是一条 `dnf install docker-io` 命令。

io 后缀可能看起来有一点神秘，但是 Docker 这个名字已经被一个实现桌面功能的包占了，所以只好用 `docker-io` 来代替。

2. 接下来，需要运行 docker 服务（译者注：以下命令依赖 systemd 系统，CentOS 7 之后的版本才默认支持）：

```
sh systemctl enable docker
systemctl start docker
```

3. 我们需要另一个工具，Docker Compose，在本书撰写的时候，还没有 Fedora 的包。如果在你的包库里找不到，参照这个网页的说明 `https://docs.docker.com/compose/install/`。

Docker Compose 用来自动启动数个 Docker 应用，比如我们在 GitLab 的例子里需要的一个数据库服务器和一个 web 服务器。

Gerrit

一个基本的 Git 服务器已经足以胜任许多用途了。

尽管如此，有些时候你需要精确地控制工作流。

一个实际的例子是把变更合并到关键的基础设施配置代码里。虽然我认为 DevOps 不应该对基础设施代码有不必要的规程，但是不可否认确实在某些时候它还是有用的。否则，开发者可能会对提交基础设施的变更感到紧张，并希望更有经验的人能一起审查代码。

Gerrit 是基于 Git 的代码审查工具，它可以提供一个这种状况下的解决方案。简而言之，Gerrit 可以让你创建规则来允许开发者审查和批准其他开发者对代码库的变更。由资深的开发者审查经验不足的开发者的变更，通常来说多双眼睛关注代码会带来更好的质量。

Gerrit 基于 Java 并在后台使用了基于 Java 的 Git 实现。

Gerrit 可以作为 Java 的 WAR 文件来下载，并且提供了一个集成创建方法。它需要依赖一个关系型数据库，但是选择一个基于 Java 的 H2 嵌入式数据库就足以评估 Gerrit 了。

更简单的方法是使用 Docker 来尝试 Gerrit。在 Docker hub 上有数个 Gerrit 镜像可供选择。我们为这次评估的练习选择了下面这个：`https://hub.docker.com/r/openfrontier/gerrit/`。

为了用 Docker 运行一个 Gerrit 实例，采取以下步骤：

1. 初始化并启动 Gerrit:

```
sh docker run -d -p 8080:8080 -p 29418:29418 openfrontier/gerrit
```

2. 打开浏览器并访问 `http://<docker host url>:8080`
   现在，我们可以尝试想要的代码审查功能了。

安装 git-review 包

在本地环境安装 `git-review`:

```
sudo dnf install git-review
```

这条命令将安装一个 Git 的帮助程序来与 Gerrit 交互。它增加了一个新命令，

`git-review`，用来替代 `git push` 把变更推送到 Gerrit Git 服务器上去。

历史修正主义的价值

当我们和团队的其他成员一起工作在相同代码上时，代码的历史就变得比自己工作更重要了。文件变更的历史记录成为了一种交流的方式。在使用诸如 Gerrit 之类的代码审查工具审查代码时尤为重要。

代码变更也需要容易理解。因此，虽然可能反直觉，为了让历史更清晰而编辑历史是很有帮助的。

举个例子，考虑这样的场景：当你做了一系列的变更而后改变主意要删掉它们。生成和删除的这些变更对其他人来说没有用。另一个场景是当你有一系列的小提交时，如果把它们作为一个提交会更加容易令人理解。以这种方式合并提交在 Git 文档里称为**挤压**（**squashing**）。

另一种让历史变得复杂的事例是，你在上游的中央库里合并了许多次，而合并的提交被加进了历史。在这种情况下，我们首先想要通过移除本地变更来简化变更，然后获取并应用上游库的变更，最后再重新应用我们的本地变更。这个流程被称为**变基**（**rebasing**）。

Gerrit 适用于 squashing 和 rebasing。

变更应该整洁，最好只是一个提交。这并不是 Gerrit 的特殊要求，优雅地打包能让一个审查者更容易理解你的变更。审查将会基于这个提交。

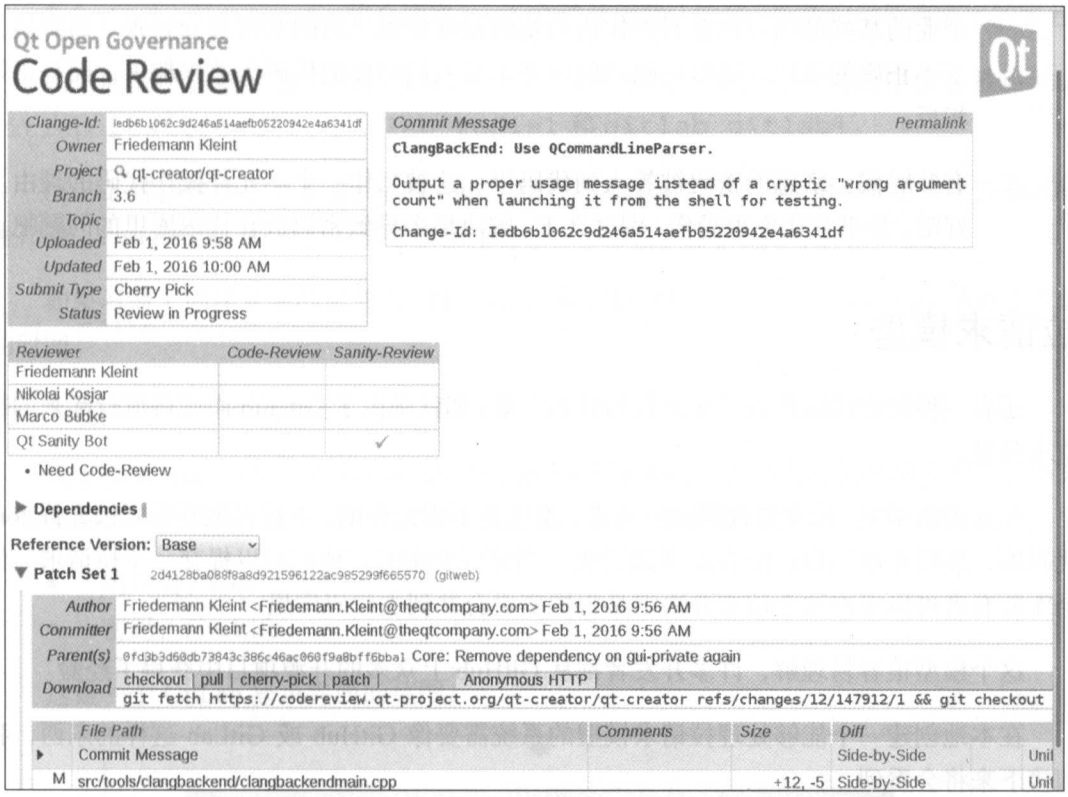
1. 刚开始时，需要从 Git/Gerrit 服务器端获取最新的变更。我们把自己的变更变基到服务器端的顶部：

```
sh git pull --rebase origin master
```

2. 然后挤压本地的提交：

```
sh git rebase -i origin/master
```

现在，看一看 Gerrit 的 web 界面：



我们现在可以批准变更，将它合并到主干分支里。

Gerrit 有许多值得探索的内容，但是作为评估的基础，这些最基本的规范应该就已经足够了。

花这么大的力气得到这样的结果是否值得？我的观察如下：

- Gerrit 允许对敏感的代码库进行细粒度的操作。变更可以在授权人审查之后入库。

  这是 Gerrit 最主要的优势。别连原因都不知道就莫名其妙地强制代码审查。只有人人都参与其中，才会获得明显的效益。最好约定其他的非正式代码审查方式而不是一个以力服人的系统。

- 如果 Gerrit 的替代方案完全不允许操作代码库或者只能是只读操作，那还是用 Gerrit 吧。

> 企业的某些部门可能会对操作诸如基础设施配置之类的东西过分紧张。通常是为了不正确的原因。经常面临的问题并不是人们对你的代码太感兴趣了，而是正好相反。
>
> 有些时候，敏感的密码被签入到代码里，这被当作一个不允许操作代码的理由。好吧，如果它让你很受伤，别这么干。解决那个导致密码放在代码库里的问题吧。

拉请求模型

还有一种解决代码审查工作流程问题的方案：那就是由于 GitHub 而变得流行起来的拉请求模型。

在这种模型里，除非是代码库所有者，推送是不被允许的。不过其他开发者被允许 fork 代码库，然后在他们自己的 fork 里做变更。当完成变更时，他们可以提交一个拉请求。代码库所有者可以审查这个请求并决定是否把变更合并到主代码库里。

这个模型很容易理解，许多开发者都从 GitHub 上众多的开源项目中获得了经验。

在本地创建一个能够处理拉请求模型的系统需要像 GitHub 或 GitLab 这样的东西，我们接下来将会看到。

GitLab

GitLab 在 Git 之上支持许多方便的功能。它是基于 Ruby 的又大又复杂的软件系统。因此，由于需要获取所有正确的依赖使得它难以安装。

在 `https://registry.hub.docker.com/u/sameersbn/gitlab/`上有一个很不错的 GitLab Docker Compose 文件。如果你紧随前文 Docker 的说明，包括安装 `docker-compose`，现在启动一个本地的 GitLab 实例会变得相当容易：

```
mkdir gitlab
cd gitla
wget https://raw.githubusercontent.com/sameersbn/docker-gitlab/master/
docker-compose.yml
docker-compose up
```

命令 `docker-compose` 将会读取 `.yml` 文件并用默认的演示配置启动所有需要的服务。

如果阅读了控制台窗口的启动日志，你会注意到三个单独的应用程序容器被启动：`gitlab postgresql1`、`gitlab redis1` 和 `gitlab gitlab1`。

GitLab 容器包含了基于 Ruby 的 web 应用程序和 Git 后端功能。Redis 是一个分布式键值存储，PostgreSQL 是一个关系型数据库。

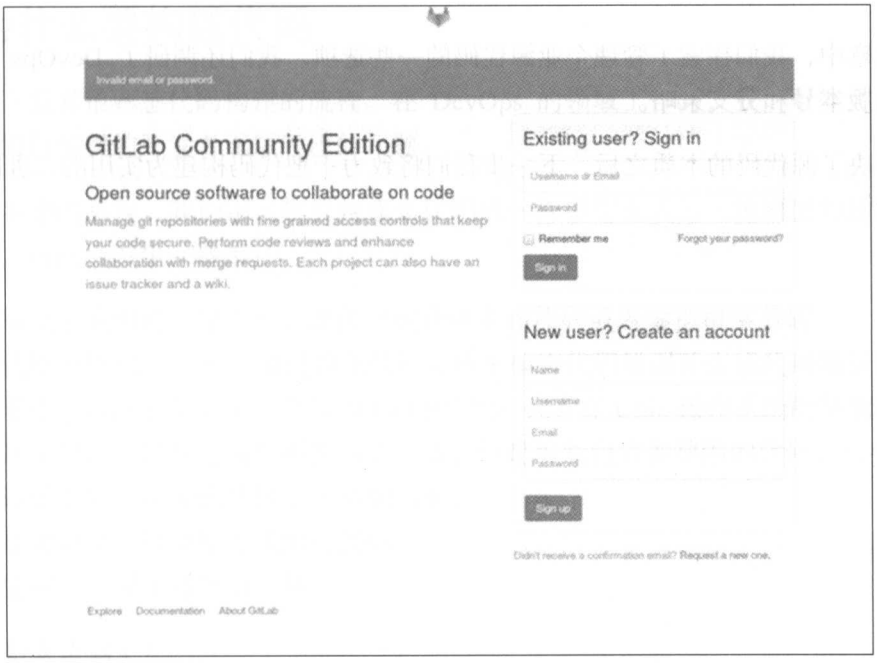
如果习惯了创建复杂的服务器功能，你会感激我们的 `docker-compose` 节省了大量的时间。

文件 `docker-compose.yml` 在 `/srv/docker/gitlab` 创建了数据卷。

为了登录 web 界面，使用 GitLab Docker 镜像安装说明里提供的管理员密码。以下是一份复制，但是请注意 Docker 镜像作者可能会在适当的时候改变：

- 用户名：root。
- 密码：5iveL!fe。

这是一个 GitLab 登录界面的截图：



尝试从 GitHub 或者本地导入一个项目到你的 GitLab 服务器。

看一看 GitLab 如何可视化提交的历史记录和分支。

在探索 GitLab 时，你也许会同意它确实提供了大量有意思的功能。

当评估功能时，注意它们是否真的可能被用上。GitLab 或者类似的软件，能解决你的什么重要问题？

事实证明 GitLab 增加的主要价值就像下面两个功能示例那样，能消除 DevOps 工作流程的瓶颈：

- 用户 ssh 密钥的管理。
- 新版本库的创建。

这些功能通常被认为是最有用的。

可视化功能也非常有用，但是 Git 客户端的可视化对开发者来说更加实用。

总结

在本章中，我们探索了管理企业源代码的一些选项。我们还调研了 DevOps 内部的决策制定、版本号和分支策略。

在解决了源代码的本质之后，下一步我们将致力于把代码构建为实用的二进制工件。

5
构建代码

你需要一个系统来构建代码，并且还得在某个地方构建它。

Jenkins 是一个灵活的开源构建服务器，可以满足你的任何需求。我们也会探索一些 Jenkins 的替代方案。

我们还将探索不同的构建系统以及它们如何影响 DevOps 的工作。

我们为什么要构建代码

许多开发者很熟悉代码构建的流程。在 DevOps 的领域工作时，我们可能会面临一些专攻特定组件编程的开发者无须经历的问题。

根据本书的宗旨，我们定义软件构建是代码从一种模型变成另一种模型的过程。在这个过程中，可能会发生一些事：

- 取决于我们的产品平台，源代码编译成本地代码或者虚拟机字节码。
- 代码分析（Linting）：通过静态代码分析来检查代码错误并生成代码质量度量值。术语“Linting”来源于一个称为 Lint 的程序，包含在 Unix 操作系统的早期版本里。这个程序的目标是找到那些语法正确，但是可能包含着缺陷的代码，这些代码可以被不同于编译的其他工序鉴别出来。
- 单元测试，以可控方式运行代码。
- 生成可以用于部署的产品。

这真是太苛刻了！

并不是所有的代码都会通过每一个阶段。例如解释型语言可能就不需要编译，但是它们可以从质量检查中受益。

构建系统的各个方面

在软件开发的历史上，许多种构建系统逐步发展。有时可能会让人觉得构建系统的数量比编程语言还多。

这里有一个简短的列表，你自己感觉一下数量会有多少：

- 对于 Java 来说，有 Maven、Gradle 和 Ant。
- 对于 C 和 C++来说，有各种不同的 Make。
- 对 Clojure 这个 JVM 的语言来说，有 Leiningen、Boot 和 Maven。
- 对于 JavaScript 来说，有 Grunt。
- 对于 Scala 来说，有 sbt。
- 对于 Ruby 来说，有 Rake。
- 最后，当然我们还有各种各样的 shell 脚本。

视企业的大小和构建产品的类型而定，你可能会碰到若干个这样的工具。为了让生活更有乐趣，各个企业发明专属的构建工具也是很常见的事。

作为对许多构建工具复杂性的回应，常用的点子是将特定工具标准化。如果你构建的是复杂的异构系统，效率不会高。例如，用 Grunt 来构建 JavaScript 就是比用 Maven 或者 Make 来得简单，而用 Maven 来构建 C 语言就不是很有效率，等等。一般来说，工具存在即合理。

通常，企业标准化一个单独的生态系统，比如 Java 和 Maven 或者 Ruby 和 Rake。除此之外的其他构建系统主要用来处理本地组件和第三方组件。

总而言之，我们不能假设在自己企业的代码库里只会碰到一个构建系统，就像我们也不能假设只用一种编程语言那样。

我发现这条规则在实践中很有用：一个开发者签出代码之后，应该能够在他或她的本地开发机上顺利地构建。

这暗示了我们应该标准化版本控制系统，并且有一个单独的接口来开始本地构建。

如果需要支持不止一个的构建系统，就基本上意味着你需要把一个构建系统包装成另一个。构建的复杂性会因此而被隐藏，并同时允许不止一个的构建系统。对某个特定构建不熟悉的开发者也能有望签出代码并且比较轻松地构建。

例如 Maven 适用于声明式 Java 构建。Maven 也能够从自己的构建内部开始其他的构建。

通过这种方式，在以 Java 为中心的企业内的开发者可以期待以下命令总能构建企业的一个组件：

```
mvn clean install
```

一个实际的例子是用 Nullsoft NSIS Windows 安装系统来创建一个 Java 桌面应用安装工具。Java 组件用 Maven 来构建。当 Java 工件准备好了以后，Maven 调用 NSIS 安装器脚本来生成一个自包含的可执行文件，可用于 Windows 安装。

虽然 Java 桌面应用近来不那么时髦了，但是在某些领域内它们还是用得很广。

Jenkins 构建服务器

一个构建服务器，本质上是一个基于各种触发器构建软件的系统。有许多构建服务器可供选择。本书中，我们将看一看 Jenkins，一个用 Java 编写的很流行的构建服务器。

Jenkins 是 Hudson 构建服务器的一个 fork。Kohsuke Kawaguchi 曾经是 Hudson 的主要贡献者，在 2010 年 Oracle 获得了 Hudson 的注册商标之后，他继续工作在 Jenkins fork 上。如今在这两个分支中，Jenkins 显然更加成功。

Jenkins 对构建 Java 代码有特别的支持，但是绝不只限于构建 Java。

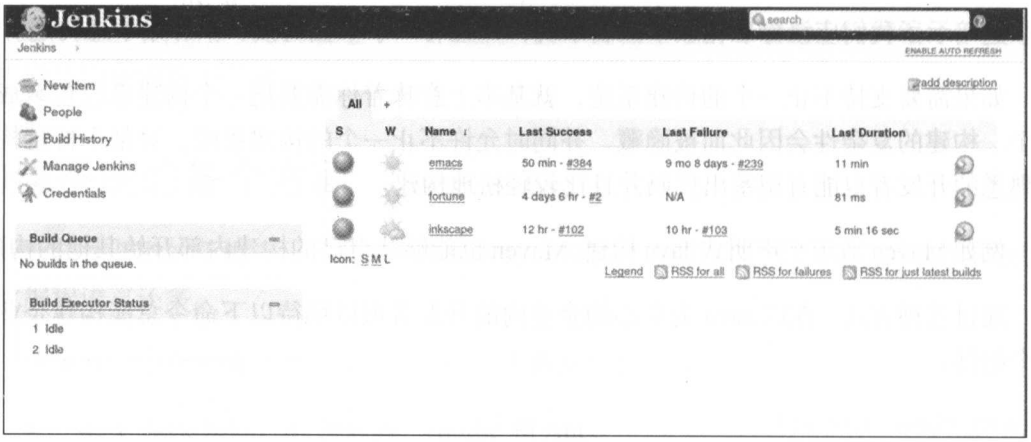
新建一个基础 Jenkins 服务器并不算难。在 Fedora 上，可以通过 `dnf` 来安装：

```
dnf install jenkins
```

通过 `systemd` 将 Jenkins 作为服务管理：

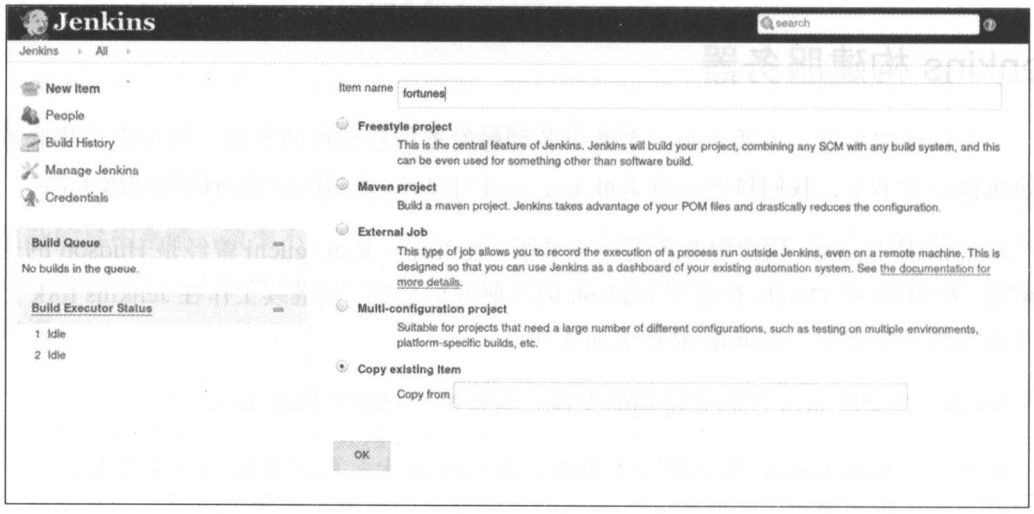
```
systemctl start jenkins
```

现在可以看一看 `http://localhost:8080` 的 web 界面：



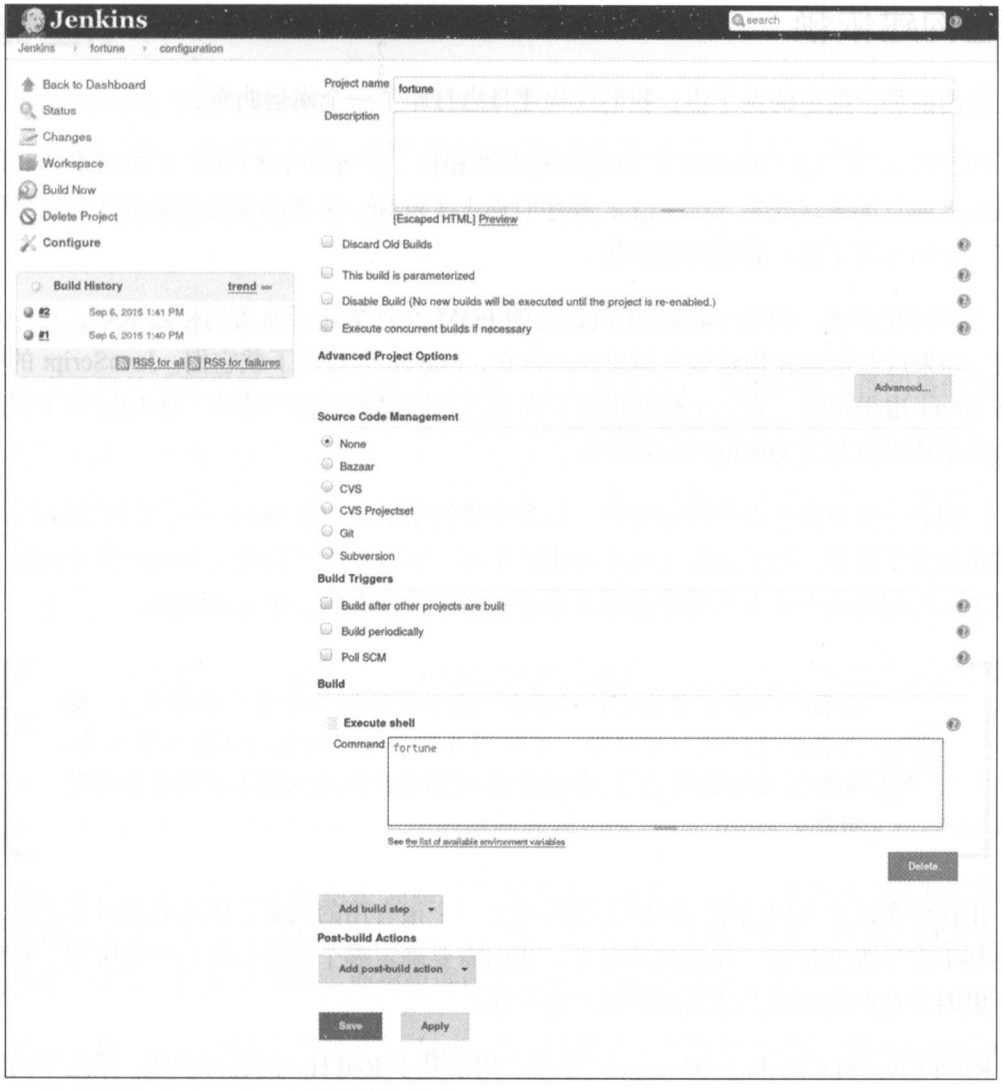
截图上的 Jenkins 实例上已经有了几个定义过的任务(job)。Jenkins 的基本实体是任务，它有几种可选的类型。让我们用 web 界面创建一个简单的任务。为了保持简单，这个任务只会打印一个经典的 Unix 幸运符：

1. 创建一个类型为 Freestyle project 的任务：



2. 添加一个 **shell build** 的步骤（ step)。

3. 在 shell 的命令行（ Command ）里，输入 `fortune`：



不管何时运行这个任务，一个 `fortune` 的引用将会被输出到任务的日志里。

任务可以手动启动，然后你就会找到一个任务执行的历史记录，还可以查看每次执行的日志。它保存了以前执行的历史记录，当你试图找出导致构建失败的那个变更并将其修复时非常好用。

如果缺少 `fortune` 命令，用 `dnf install fortune-mod` 来安装，或者更加简单地运行 `date` 命令来代替。这仅会在构建日志里打印日期而不是经典的连珠妙语。

管理构建依赖

在前面那个简单的例子里，我们往构建日志打印了一个幸运曲奇。

这次练习还不能与管理真正构建的复杂性相比，但至少我们学会了如何安装和启动 Jenkins，而且如果在安装 `fortune` 程序时遇上了麻烦，你还会对持续集成服务器的阴暗面有一个初步的了解：管理构建依赖。

有些构建系统，例如 Maven 工具，它的 POM 文件描述了所需的构建依赖，这一点很不错。如果这些依赖在构建服务器里还不存在，Maven 会自动下载它们。JavaScript 的构建工具 Grunt 也很类似，有一个构建描述文件包含了构建所需要的依赖。Golang 为了完成构建，甚至还可以包含 GitHub 库的链接。

C 和 C++构建存在着不同的挑战。许多项目使用 GNU 的 Autotools，它的 Autoconf 并不描述需要的依赖，而是适配主机上可用的依赖。所以，为了构建文本编辑器 Emacs，你需要先运行配置脚本来查明构建系统上潜在的诸多依赖中哪一个是可用的。

如果一个非必需的依赖不存在，比如支持图片所需的的图片库，那么在最终可执行文件里，支持图片这个可选功能将不能正常工作。你仍然可以构建程序，但是得不到那些在构建机器上没有准备好的功能。

让软件在许多不同的系统配置上都可用是一个很有用的功能，因为我们并不总是想让构建基于重量级的配置。在这种情况下，我们需要确定哪个功能百分百终将可用。我们当然不想因为在产品环境上遗漏功能而大吃一惊。

RPM（Red Hat Package Manager）系统，用于基于 Red Hat 的各种系统，给这个问题提供了一个解决方案。RPM 系统的核心是一个称为 spec 文件的构件描述文件。它列出了需要成功构建所需的构建依赖、构建命令，还有使用的配置选项。因为一个 spec 文件本质上是一个基于宏的 shell 脚本，所以你可以用它来构建许多类型的软件。RPM 系统也认为构建源代码应该从零开始。可以用构建之前通过 spec 文件给源代码打包的方式适配源代码。

最终工件

在使用 RPM 系统完成构建之后，你得到了一个 RPM 文件，这种类型可以很方便地给 Red Hat 系列操作系统部署工件。对于 Debian 系的发行版来说，你得到的是个 `.deb` 文件。

Maven 构建的最终结果一般是企业级档案，简称 EAR 文件。它包含着 Java 企业级应用。

它是我们之后要部署到生产服务器上的最终部署工件。

本章我们关心部署所需工件的构建，在第 7 章部署代码中，我们讨论工件的最终部署。

尽管如此，即使在构建工件的时候，我们也需要理解如何部署它们。现在，我们将要使用以下的经验法则：操作系统级别的包优于特定的包。这是我的个人喜好，其他人可能不同意。

让我们简要地讨论一下这个经验法则的背景和其他的方案。

作为一个实际的例子，让我们考虑 Java EAR 的部署。通常，我们有几种方法来实现。这里有一些例子：

- 在基本的操作系统上通过可用的机制和渠道，用 RPM 包的方式部署 EAR 文件。
- 在 Java 应用服务器上通过可用的机制部署 EAR，例如 JBoss、WildFly 和 Glassfish。

粗看上去，通过特定的机制在 Java 应用服务器上部署 EAR 文件更好，因为它基于特定的应用服务器。如果你只部署过 Java，这可能是个合情合理的想法。然而，因为无论如何也得管理最底层的操作系统，而你也许已经有可以重用的部署方法了。

还有，因为你很可能并不只开发 Java，至少还会部署和管理 HTML 和 JavaScript，使用一个通用的部署方法开始变得更有意义。

我经历过的几乎所有企业都有包含许多不同技术的复杂架构，这个经验法则在大多数场景里都很管用。

唯一的例外是当 Unix 服务器和 Windows 服务器并存的混合环境。在这些案例里，Unix 服务器通常使用它们喜欢的包发布方式，而 Windows 服务器不得不使用自制方案蹒跚而行。这只是一个观察结果，并不表示我们会姑息这种情形。

用 FPM 取巧

构建像含有 spec 文件的 RPM 这样的操作系统包是非常有用的知识。不过，有些时候并不需要像真正的 spec 文件那么严格。毕竟 sepc 文件优化的是你的场景而非代码库创始者的场景。

有一个基于 Ruby 的名为 FPM 的工具可以直接从命令行生成适合构建的源代码 RPM 包。

可以在 GitHub 上获得这个工具：`https://github.com/jordansissel/fpm`。

在 Fedora 上可以这样安装 FPM：

```
yum install rubygems
yum install ruby
yum install ruby-devel gcc
gem install fpm
```

这样就可以安装一个包装了 FPM Ruby 程序的 shell 脚本。

FPM 有意思的一个地方是它可以生成不同类型的包，其中就有 RPM 和 Debian 所支持的类型。

这里有一个制作“Hello world”的 shell 脚本的简单例子：

```
#!/bin/sh
echo 'Hello World!'
```

我们想要把 shell 脚本安装在`/usr/local/bin` 里，所以在主目录下创建一个如下结构的目录：

```
$HOME/hello/usr/local/bin/hello.sh
```

设置脚本可执行，然后打包：

```
chmod a+x usr/local/bin/hello.sh
fpm -s dir -t rpm -n hello-world -v 1 -C installdir usr
```

这样的结果是一个名为 hello-world、版本为 1 的 RPM 包。

为了测试这个包，我们首先列出内容，然后安装：

```
rpm -qivp hello-world.rpm
rpm -ivh hello-world.rpm
```

这个 shell 脚本现在应该被顺利安装到/usr/local/bin 里了。

FPM 是创建 RPM、Debian 还有其他种类包的一个非常便利的方法。有点投机取巧的感觉!

持续集成

使用构建服务器最主要的好处是实现持续集成。每当检测到代码库的变更，一次测试新提交的代码的构建就开始了。

由于可能有许多开发者在代码库工作，每个人使用的版本都略有区别，所以知道所有不同的变更能否一起正确工作就显得非常重要。这被称为**集成测试**。如果集成测试还遥遥无期，就存在一个不同代码分支各自演进，合并变得不再容易的问题。这个结果通常被称为“合并地狱”。因为各个分支大不相同，如何把开发者的本地变更合并到主干上也变得不再清晰。这种情形不是我们想要的。合并地狱的根源经常可能令人大吃一惊：就是心理问题。为了把你的变更合并到主线，有一个心理障碍需要克服。DevOps 工作的一部分就是简化事情，降低像提交变更这种重要工作的感知成本。

持续集成上的构建通常比开发者的本地构建更加严格。这些构建需要很长的时间来运行，但是如今的高性能硬件已不再昂贵，我们的构建服务器足以应付。

如果构建快到不至于让人感到无聊，开发者将会对频繁提交充满热情，集成问题将会更早出现。

持续交付

在持续集成这一步胜利完成之后，你拥有的崭新工件就可以部署到服务器上了。一般来说，有一些测试环境会设置得像产品服务器那样。

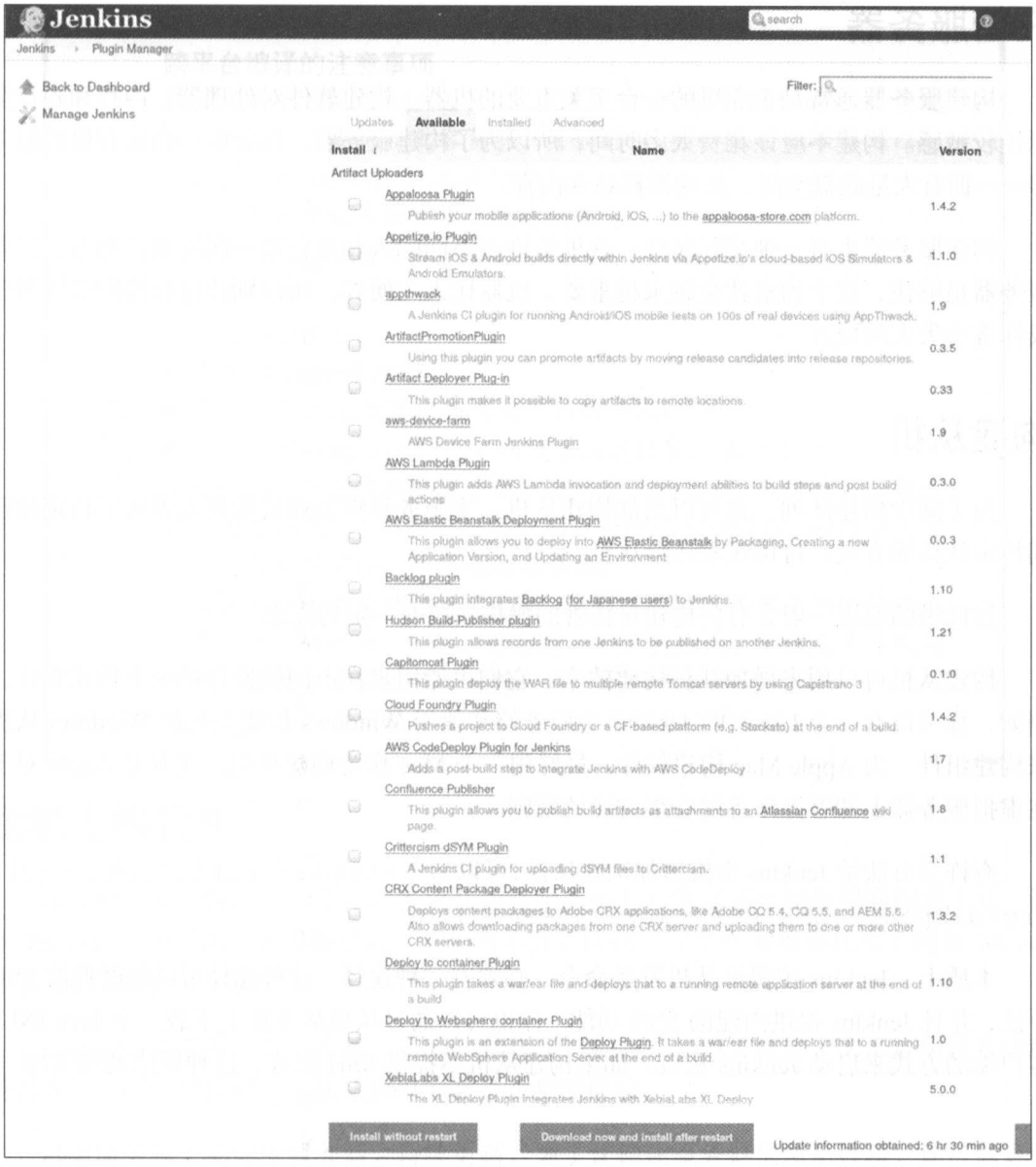
本书后面将会讨论部署系统的选择。

一般来说构建服务器所做的最后一步，是从成功的构建里把最终工件部署到工件库里。在那里，部署服务器接管了部署到应用服务器的责任。在 Java 世界里，Nexus 库管理器相当常见。它支持的格式不仅仅是 Java，还有例如 JavaScript 工件和 Yum 体系的 RPM。Nexus 现在也支持 Docker Registry 的 API 了。

为 RPM 发行版使用 Nexus 只是可选方案之一。你可以用 shell 脚本很轻松地构建 Yum 体系。

Jenkins 插件

Jenkins 有一个可以给构建服务器增加功能的插件系统。在 Jenkins 的 web 界面上，可以选择安装许多不同的插件。大部分安装都不需要重启 Jenkins。这个截图展示了一些可选的插件：



其中，我们需要 Git 插件来轮询源代码库。

范例企业决定构建 Clojure，所以我们将要安装 Leingingen 插件。

托管服务器

构建服务器通常是企业里的一台至关重要的机器。构建软件对处理器、内存和磁盘资源比较敏感。构建不应该花费太多时间，所以为了构建服务器，你需要一台高规格的服务器——拥有大量磁盘空间、处理器核数和内存。

构建服务器也有一种交际属性：这里是许多不同的人和角色第一次集成的地方。只要服务器足够快，这个因素就会越来越重要。机器比人工便宜，所以别让这台特别的机器变成你贪小失大的地方。

构建从机

为了减少构建队列，你可以增加构建从机。主服务器将会通过轮询或者绑定特定构建到特定从机的方式，将构建发送给从机。

这样做的原因一般是有些构建对托管的操作系统有一定的需求。

构建从机可以用来增加并行构建效率。它们也能用来在不同的操作系统上构建软件。例如，你可以有一个 Linux 的 Jenkins 主服务器和使用 Windows 构建工具的 Windows 从机来构建组件。为 Apple Mac 构建软件，最好用一个 Mac 作为构建从机，尤其是 Apple 对于在虚拟服务器上部署操作系统有很古怪的规则。

有许多方法给 Jenkins 主机增加构建从机，可以参考 `https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds`。

本质上，Jenkins 主机给从机发送命令，必须有一种途径。这种途径可以是经典的 SSH 方法，并且 Jenkins 提供内建的 SSH 功能。你也可以通过从机从主机上下载一个 Java JNLP 客户端的方式来启动 Jenkins 从机。如果构建从机不提供 SSH 服务，这种做法就很实用。

跨平台编译的注意事项

尽管可以使用 Windows 构建从机，有些时候用 Linux 来构建 Windows 软件要来得更容易些。诸如 GCC 那样的 C 编译器可以通过配置使用 MinGW 包来进行跨平台编译。

是否这样做更容易取决于要构建的软件。

一个大系统通常由许多不同的部分组成，一些部分可能包含了不同平台的本地代码。

这儿有几个例子：

- 本地的 android 组件。
- 本地为了性能而用 C 语言编写的服务器组件。
- 本地为了性能而用 C 或 C++语言编写的客户端组件。

本地代码的多少有些取决于你所在的企业的性质。电信产品一般有许多的本地代码，例如视频编解码器和硬件接口代码。银行系统可能有用本地代码编写的高速消息系统。

这样做的一方面是件很重要的事：能够在构建服务器上便捷地构建所有使用中的代码。否则，就会有一种不良倾向，那就是某些代码只能在几台吃灰的机器上构建。这是我们需要避免的风险。

企业里的系统需要的究竟是什么，只有你自己才知道。

主机上的软件

视构建的复杂性而定，你可能需要在构建服务器上安装许多不同类型的构建工具。记住 Jenkins 主要是用来触发构建的，它们并不自己构建。这个任务被委托给了例如 Maven 或 Make 这样的构建系统。

以我的经验来看，有一个基于 Linux 操作系统的主机是最方便的。大多数构建系统都可以在发行版库里找到，直接从那里安装是非常简便的。

部署服务器上的应用服务器会持续更新，为了让构建服务器也总是最新，你可以直接使用同一台部署服务器来做构建。

触发器

你可以设置一个定时器来触发构建，或者轮询代码库直到发生变更时才构建。

可以同时使用这两种方法：

- 最常用的是轮询 Git 库，这样每次提交都会触发构建。
- 可以触发比持续构建更久也更加彻底的每夜构建。由于这些构建发生在假定无人工作的夜晚，慢一点也关系不大。
- 一个上游的构建可以触发一个下游的构建。

你还能让一个任务里的成功构建触发另一个任务。

任务链和构建流水线

能把任务串起来通常都是非常有用的。最简单的方式是：当第一个任务成功完成时，会通过事件触发第二个任务。几个任务可以用这种方式在链中传递。这样的一个构建链一般足以胜任大多数工作。有时构建步骤的可视化和细粒度控制任务链都是我们想要的。

在 Jenkins 术语里，链上的第一个构建被称为上游构建，第二个被称为下游构建。

虽然这种链式构建的方式一般够用，但还是有可能需要更好地控制构建链。这样的构建链通常被称为流水线或工作流。

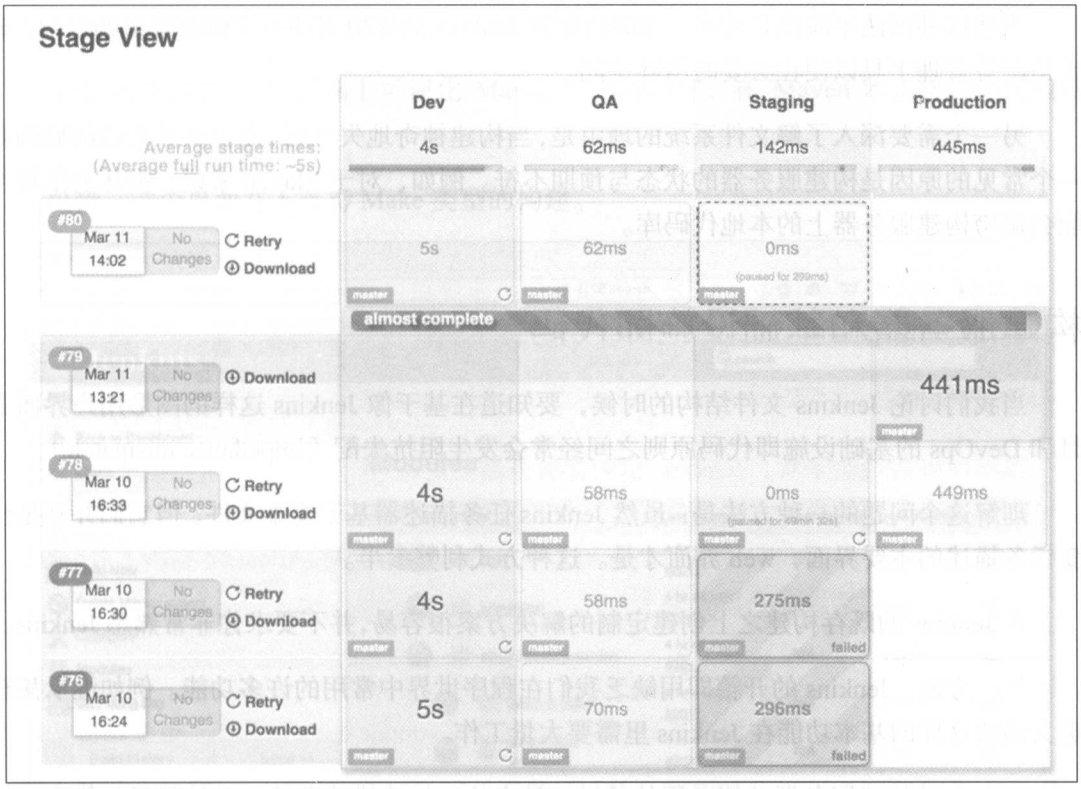
有许多为 Jenkins 创建更好流水线的插件，它们的数量表明了确实有改进这方面的大量需求。

多任务插件和工作流插件是其中的两个例子。

工作流插件更加先进，它有一个优势是可以用 Groovy DSL 来描述而不是在 web 界面上改来改去。

工作流插件是 CloudBees 主推的，它现在是 Jenkins 最主要的贡献者。

工作流的一个例子如下：



当看到工作流插件使用的 Groovy 构建脚本时，你可能会感到 Jenkins 只是一个有着 web 界面的构建工具而已，这种想法多少有点道理。

Jenkins 文件系统结构概览

了解构建如何最终反映到文件系统中一般还是挺有帮助的。

用 Fedora 包的话，Jenkins 任务存储在`/var/lib/jenkins/jobs`。

每一个任务都有自己的目录，任务描述 XML 文件和称为工作区（workspace）的构建用目录就存储在这个目录里。任务 XML 文件可以备份到另一台服务器上，以便能够在灾难性故障后重建 Jenkins 服务器。专用的备份插件也是一个可选方案。

构建会消耗掉大量的空间，所以你有时候可能需要手动清理空间。

当然这种做法不应该是常态。你应该配置 Jenkins 只保留支持一定构建数量的空间，或者是配置管理工具以便在必要时清除空间。

另一个需要深入了解文件系统的理由是，当构建神奇地失败时，需要调试失败的原因。一个常见的原因是构建服务器的状态与预期不符。例如，对于 Maven 来说，损坏的依赖可能会搞垮构建服务器上的本地代码库。

构建服务器和基础设施即代码

当我们讨论 Jenkins 文件结构的时候，要知道在基于像 Jenkins 这样的图形用户界面工具和 DevOps 的基础设施即代码原则之间经常会发生阻抗失配（impedance mismatch）。

理解这个问题的一种方法是：虽然 Jenkins 任务描述器基于文本文件，但它们并不是改变任务描述的主要界面，web 界面才是。这种方式利弊参半。

在 Jenkins 的既存构建之上创建定制的解决方案很容易，并不要求你非常熟悉 Jenkins。

另一方面，Jenkins 的开箱即用缺乏我们在程序世界中常用的许多功能。例如继承甚至定义函数这样的基本功能在 Jenkins 里需要大量工作。

GitLab 的构建服务器功能，用的是另一种方法。构建描述器从一开始就只是代码。如果并不需要 Jenkins 提供的所有能力，GitLab 的这个特性值得你一看。

按依赖顺序构建

因为构建的一部分可能会依赖于其他部分，许多构建工具都有构建树的概念：为完成构建而有顺序地构建依赖。

在 Make 类型的工具里，它被显式地描述。例如这样：

```
a.out : b.o c.o
b.o : b.c
c.o : c.c
```

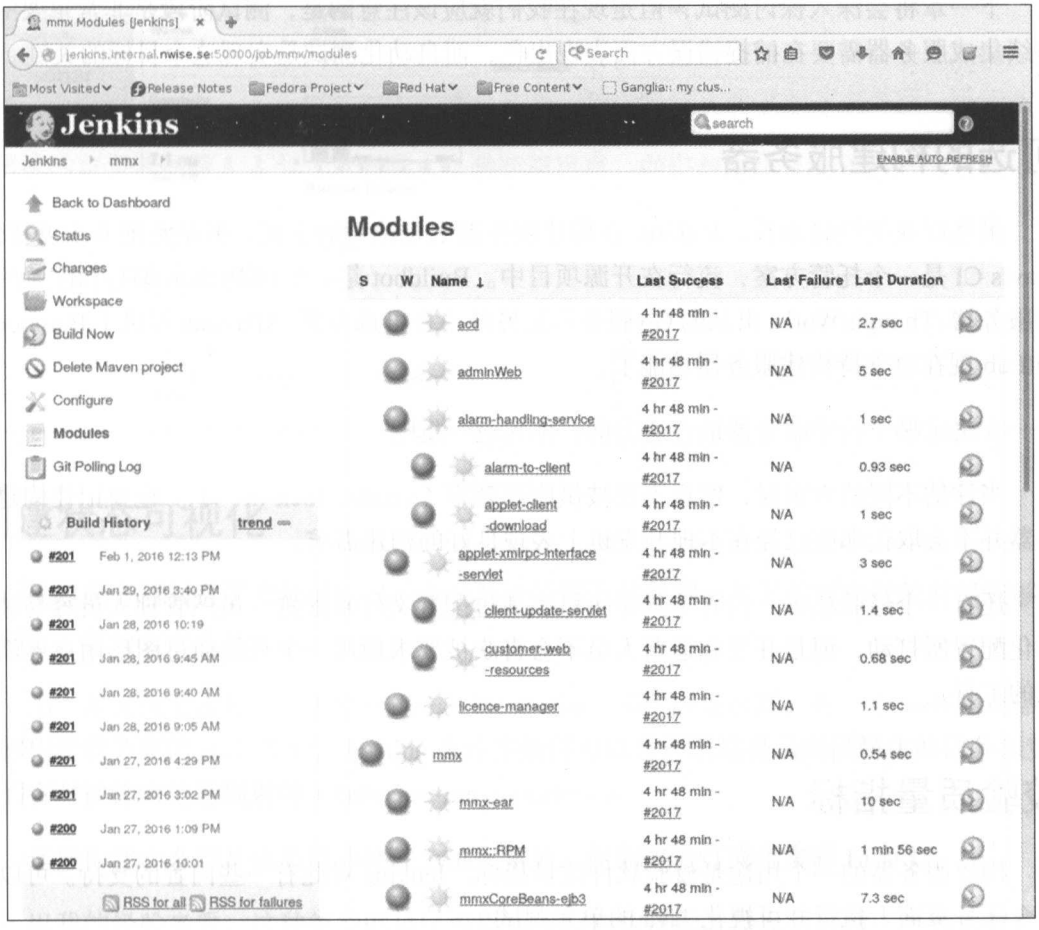
因此，为了构建 `a.out`，必须先构建 `b.o` 和 `c.o`。

在像 Maven 这样的工具里，构件图来源于由我们为工件设置的依赖。另一个 Java 构建

工具 Gradle，也会在构建之前先创建一个构件图。

Jenkins 支持在 web 界面上可视化 Maven 的构建顺序，在 Maven 术语里称为**反应器**（**reactor**）。

可惜，这个界面并不支持 Make 类型的构建。



构建阶段

Maven 构建工具的主要优势就是它把构建流程标准化了。

这一点对大型企业来说非常有帮助，因为它不需要再发明自己的构建标准了。其他的

构建工具实现各种构建流程一般更加随意。Maven 的严苛有好有坏。有时，刚开始用 Maven 的人们会怀念像 Ant 那样工具所带来的自由。

你可以用任何工具来实现这些构建，但是当工具本身不强迫构建、测试和部署的标准顺序时，很难还能保持习惯。

下一章将会深入探讨测试，但是现在我们就应该注意的是，测试阶段是非常重要的。持续集成服务器需要在捕捉错误方面表现出色，而自动化测试是实现这个目标的关键。

可选的构建服务器

虽然以我的经验来看，Jenkins 在构建服务器上是绝对的主流，但是它绝非不可替代。Travis CI 是一个托管方案，流行在开源项目中。Buildbot 是一个用 Python 编写和配置的构建服务器。ThoughtWorks 出品的 Go 服务器是另外一种可选方案。Atlassian 提供了 Bamboo。GitLab 现在也支持构建服务器功能了。

在决定哪个构建服务器最合适之前，你先逛一逛吧。

当评估不同的方案时，留意不要被供应商套牢（vendor lock-in）了。也要记住构建服务器并不会取代那些已经在本地开发机上表现良好的构建需求。

还有一个经验法则：看看工具是否可以通过配置文件来配置。虽然管理人员容易被图形化配置所打动，但是开发和运营人员不会喜欢被要求使用一个只能通过图形用户界面配置的工具。

校验质量指标

构建服务器的一个用途是校验软件质量指标。Jenkins 对此有一些内置的支持。可以在一个任务页面上执行并可视化 Java 的单元测试。

另一个更高级的可选方案是使用 Sonar 代码质量可视化器，参见下图。Sonar 测试在构建阶段运行并传送到 Sonar 服务器上，在那里进行存储和可视化。

一台 Sonar 服务器是一个让开发团队看到他们努力改良代码库的成果的好办法。

Sonar 服务器的缺点是有时会减缓构建速度。建议每天夜里运行一次 Sonar 构建。

构建状态可视化

构建服务器制造了大量可以显示在公共显示器上的数据。若是构建失败的时候能够立即察觉，就会有很大帮助。

第一步仅仅是连接一个类似资讯站（kiosk-like）那样的显示器，有一个 web 浏览器指向你构建服务器的 web 界面。Jenkins 有许多插件可以为资讯站显示提供简化的任务概览。它们有时被称为**信息辐射体（Information radiators）**。

连接构建状态到其他类型的硬件上也很常见，例如熔岩灯或者多彩 LED 灯。

以我的经验来看，这种类型的显示可以让人们对构建服务器充满兴趣。尽管保持长期使用比开始拥有更加困难。如果你想把屏幕放到一个不容易被看到的地方以避免分心，那么做这件事就变得毫无意义了。

慎重放置熔岩灯和屏幕的组合非常有帮助。熔岩灯并不常亮，所以没那么吸引人。当一个构建失败发生时，它亮起来了，于是你就知道你应该看一看构建信息辐射体了。熔岩

灯甚至能够表达构建质量的历史记录。当熔岩灯亮的时候，它会发热，过一会儿，熔岩在灯里环绕。当错误被修复，灯冷却了，但是热量还会存在一小段时间，所以熔岩还会环绕一段时间，视修复构建错误的耗时而定。

严肃对待构建错误

构建服务器可以随心所欲地传递出错误和代码质量问题的信号，如果开发团队不关心这些问题，那么这些通知和可视化的投资收益为零。

这并不是技术本身可以解决的。流程需要每个人都同意，让人人都能看见自己参与所带来的利益是达成共识的最简方法。

问题是企业里总是有一大堆迫在眉睫的事。构建错误会比产品错误更重要吗？还有代码质量指标方面，如果改进代码质量需要数年，值得着手解决这个问题吗？

我们怎样解决这类问题？这里有一些想法：

- 不要过分追求代码质量指标。可以先减少测试的数量，直到代码质量报告达到了可以修复的水平。解决之后再把测试加回来。
- 定义问题的优先级。产品问题优先，然后才是构建错误。在问题被完全解决之前，不要在损坏的代码之上提交新代码。

健壮性

尽管想让构建服务器成为持续交付流水线的中心之一，但我们也要考虑当构建服务器瘫痪的时候，构建和部署的流程不应该停滞不前。为此，构建本身应该尽可能健壮，并且可以在任何主机上重复工作。

这对像 Maven 那样的一些构建来说相当容易。可即便如此，一个 Maven 构建也可能有无数的缺陷而使其无法被正常移植。

一个基于 C 语言的构建会很难移植，如果你没有幸运到所有的依赖都在操作系统库里可用的地步。还是那句话，健壮性通常能够值回票价。

总结

在本章，我们旋风般地扫过了构建代码的系统。看过了用 Jenkins 构建持续集成服务器，也检查了许多可能发生的问题，DevOps 工程师的生活总是很有意思，但并不总是很容易。

下一章我们将会继续努力，通过学习如何能够在工作流程中集成测试，来生产高质量的代码。

6
测试代码

如果需要又快又频繁地发布代码，我们就得对代码的质量有信心。因此，我们需要自动化回归测试。

本章我们会探索一些软件测试的框架，比如做单元测试的 Junit 以及用于网站前端测试的 Selenium。同时，我们也会介绍这些测试是如何在持续集成服务器上运行的，例如 Jenkins，它们组成了我们持续交付流水线的第一部分。

测试对保持软件质量非常重要，同时它自身也是一个很大的话题。

我们会在本章关注如下几个话题：

- 如何让人工测试更加简单并不易出错。
- 各种类型的测试，例如单元测试，以及如何在实践中应用。
- 自动化系统集成测试。

在上一章中，我们已经看过如何使用 Sonar 和 Jenkins 收集测试数据，在本章中我们会继续深入这个话题。

人工测试

对 DevOps 来说，即便自动化测试比人工测试带来的收益更大，人工测试依然是软件开发过程中的一个重要部分。即便没有别的原因，为了将测试自动化，我们至少也得手动

执行一次测试来验证。

接受测试（Acceptance testing）尤其难以替代，即使有人尝试去这么做。即使对那些开发功能实现需求的人来说，软件需求规范简短并且难以理解。在这些情况下，有质量保证的人盯着是有价值并不可替代的。

让人工测试变简单的东西和让自动化的集成测试变简单的东西一样，所以这两种测试策略之间也存在协同效应。

为了让质量保证人员开心，你需要：

- 管理测试数据，主要是后端数据库的内容，这样当你重复运行测试时，才能得到相同的结果。
- 为了验证缺陷是否修复，需要尽快部署新代码。

看上去显而易见，但是实践起来会有些难度。也许你的数据库太大，无法复制到测试环境。也许它包含最终用户的数据，根据法律需要保护。 在这些情况下，你需要在部署测试环境前辨识并清洗任何关于私人信息的数据。

每个企业都是不一样的，所以在这个领域很难给出除了 KISS 原则（Keep it simple, stupid）之外的普适的有用建议。

自动化测试的优缺点

当你和大家交流时，大多数人对自动化测试的前景充满热情。试想下它可能会带给我们的好处：

- 更高的软件质量。
- 对于发布按照预期工作的软件有更强的自信。
- 减少费力且单调乏味的人工测试。

一切看上去都是那么美好！

但在实践中，如果在不同企业中复杂的多层产品上花费时间，你会发现人们都在讨论自动化测试，但是你也会注意到实践中却缺乏自动化测试。这是为什么呢？

如果你仅仅是编译程序并在编译完成时部署，结果可能会很糟糕。为了让现实世界中的软件能可靠工作，软件测试是完全有必要的。人工测试对于实现持续交付来说太慢。所以，我们需要自动化测试来完成持续交付。因此，让我们进一步调查围绕自动化测试的问题领域，看看怎么做才能改善这种情况：

- 廉价的测试没什么价值。

  一个问题是，单元测试是成本很低的自动化测试类型，一般来说它比其他的测试类型带来的价值更低。单元测试仍然是一种不错的测试类型，但是人工测试可能被认为会在实践中暴露更多的 bug。可能就会感觉写单元测试没什么必要了。

- 很难去创建和自动化集成测试相关的测试支架（test cradle）。

  虽然实现单元测试的测试支架或者测试夹具（test fixture）不是很困难，但是将其变得更像产品环境会越来越困难。这可能是因为缺乏硬件资源、许可证、人力等。

- 程序的功能随时间而变化，而测试必须做出相应的调整，耗费时间和精力。

  这使得自动化测试好像只是让软件开发变得更困难而没有感觉到什么收益。

  在开发和运维关系不是很紧密，或者说非面向 DevOps 的企业中，尤其如此。如果有其他人不得不处理你那些不能正常工作的烂代码，那么对于开发人员来说写点烂代码根本无关紧要。这不是一个健康的关系。这也是 DevOps 想要解决的最核心的问题。DevOps 的方式证明了这条已重复数次的规则：帮助不同角色的人们紧密工作在一起。在像 Netflix 这样的企业中，一个敏捷团队对自己服务的成功、维护及中断负有全部责任。

- 在很多不同的构建场景下编写可靠工作的健壮测试很困难。

  这样带来的后果是，很多开发人员试图禁用本地构建的测试，这样他们就可以不受打扰地完成分配给自己的需求。因为大家不写测试，随着影响测试结果的变更增加，测试最终会失败。

  构建服务器将会发现构建错误，可惜现在没人记得住测试是如何工作的，并且可能需要几天时间来修复测试错误。测试失败时，构建会显示红色，最终人们会不再关心构建的问题。反正总会有人去修复问题。

- 写好自动化测试就是太难。

  创建好的自动化集成测试确实很难。当然它也会带来好处，因为你了解了被测系统的所有方面。

  这些都是棘手的问题，尤其是因为它们大多来自人们的观念和关系。

解决这些没有灵丹妙药，但是我建议可以采取以下的策略：

- 利用人的积极性去实现测试自动化。
- 不要设定不切实际的目标。
- 一步一步来。

单元测试

通常来说，单元测试和开发人员相关度更高。主要的原因是：根据定义，单元测试用于测试系统中与其他部分隔离、定义良好的部分。因此，它们比较容易编写和使用。

许多构建系统已经内置对单元测试的支持，不用花费太大力气就可以使用。

例如 Maven，就有这样的约定：描述如何写测试才能让构建系统找到测试、执行测试并最终准备报告结果。编写测试基本上可以归结为编写测试方法，通过在源代码加注解来标记它们。因为这些都是普通的方法，所以它们可以做任何事情，但是根据约定，应该编写测试，这样就不需要花费很大的力气去运行。如果测试代码开始需要复杂的设置和运行时依赖，我们处理的就不再是单元测试了。

这里单元测试和功能测试的差别很让人困惑。通常来说，相同的底层技术，类库会在单元测试和功能测试之间重复使用。

复用一般来说是一件好事情，因为它可以让你工作在另一个领域时，从你的专业领域中获益。尽管如此，还是会时不时地带来困扰，你必须打起精神，花费一些力气，确保你正在做正确的事情。

一般的 JUnit 和特殊的 JUnit

你需要一些东西去运行你的测试。JUnit 是一个框架，它可以让你在 Java 代码中定义和运行测试。

JUnit 属于统称为 xUnit 的测试框架家族。SUnit 是这个家族的祖先，由 Kent Beck 在 1998 年为 Smalltalk 语言而设计。

JUnit 是 Java 语言的测试框架，同样想法被移植到了其他语言，如 C#。C#对应的测试框架毫无想象力地叫作 **NUnit**。N 来自 .NET，微软软件平台的名字。

在话题继续之前，我们需要了解下面的命名法则。命名的法则不特定于 JUnit，但是我们会用 JUnit 作为例子来熟悉涉及到的定义。

- **测试运行器（Test runner）**：测试运行器运行由 xUnit 框架定义好的测试。

  JUnit 可以通过命令行去运行单元测试，Maven 使用的测试运行器叫作 Surefire。测试运行器也会同时收集和汇报测试结果。以 Surefire 为例，报告是 XML 格式的，并且可以被其他工具进一步处理，如可视化。

- **测试用例（Test case）**：测试用例是最基本类型的测试定义。

  用不同的 JUnit 版本创建测试用例存在些许不同。在早期的版本中，需要从 JUnit 基类继承，最近的版本中，只需要给测试方法添加注解。这样的方式会更好些，因为 Java 不支持多继承，而且你有可能想使用自己的继承结构而不是 JUnit 的继承结构。按照约定，Surefire 也会定位类名前缀为 `Test` 的测试类。

- **测试夹具（Test fixtures）**：测试夹具是测试用例可以依赖的一个已知状态，可以使测试具有良好定义的行为。开发者有责任去实现这些测试夹具。测试夹具有时也被称为测试上下文（test context）。

  在 JUnit 中，你通常会使用`@Before` 和`@After` 注解去定义测试夹具。`@Before`，顾名思义，是在测试用例前运行，用于准备好整个环境。同理，如果有需要，用 `@After` 来恢复成初始状态。

  有时，为了更好表意，`@Before` 和`@After` 也被命名为 **Setup** 和 **Teardown**。因为使用了注解，在上下文中方法可以有最直观的名字。

- **测试套件（Test suites）**：在测试套件中你可以将测试用例分组。测试套件通常是一组共享测试夹具的测试用例。
- **测试执行器（Test execution）**：测试执行器运行测试套件和测试用例。

  这里结合了前面的所有方面。定位测试套件和测试用例，创建相关的测试夹具，然后运行测试用例。最后，收集和整理测试结果。

- **测试结果格式化工具（Test result formatter）**：测试结果格式化工具格式化测试输出结果，方便人们阅读。JUnit 使用多种多样的格式，可以被其他和 JUnit 没有直接关联的测试框架和格式化工具使用。所以，如果你的一些测试没有使用任何的xUnit测试框架，你还是可以通过提供XML格式的测试结果，将其展示在Jenkins中。因为文件格式是 XML，如果有必要，你也可以用自己的工具去生成。
- **断言（Assertions）**：断言是 xUnit 框架的一种构造，用来确保条件被满足。如果条件没有满足，它会认为这是个错误，并且汇报这个测试错误。测试用例通常在断言失败时终止。

JUnit 有很多断言方法。下面是一些可用的例子：

- 检查两个对象是否相等：

```
ssertEquals(str1, str2);
```

- 检查一个条件是否为真：

```
assertTrue (val1 < val2);
```

- 检查一个条件是否为假：

```
assertFalse(val1 > val2);
```

一个 JUnit 的例子

Java 的构建工具对 JUnit 支持很好。这有助于使 JUnit 测试框架范例变得通用。

如果我们使用 Maven，按照约定，它会期望从下面的目录中找到测试用例：

```
/src/test/java
```

Mocking

Mocking 是指编写模拟资源去实现单元测试的实践。有时候也会使用 fake 或者 stub。

例如，一个从数据库读取并响应 JSON 结构的中间件系统，可以“mock”后端的数据库来进行单元测试。否则，单元测试需要后端数据库在线，甚至需要独占访问。这可太不方便了。

Mockito 是一个 Java 的 mock 框架，被移植到了 Python。

测试覆盖率

当你听到谈论单元测试的时候，人们会经常提到测试覆盖率。测试覆盖率是测试用例中执行的应用程序代码的百分比。

为了度量单元测试覆盖率，你需要执行测试并且跟踪执行有或没有被执行的代码。

Cobertura 就是在 Java 中度量测试覆盖率的工具。其他类似的工具包括 jcoverage 和 Clover。

Cobertura的工作原理是插装Java字节码，将自己的代码段插入到已经编译好的代码中。测试用例执行时，这些度量代码覆盖率的代码段也被执行。

通常来说，百分之百的代码覆盖率是比较理想的。实际的情况并不总是这样，而且也要权衡成本和效益比。

一个简单的反例就是下面 Java 代码中的 `getter` 方法：

```
  private int positiveValue;
  void setPositiveValue(int x){
    this.positiveValue=x;
  }

  int getPositiveValue(){
    return positiveValue;
}
```

如果我们为这个方法写一个测试用例，将会获得更高的测试覆盖率。但是从另一个角度来说，这对我们来说并没有什么实际的意义。我们真正测试的是仅仅是 Java 的实现没有错误。如果 `setter` 方法包含了值是否为负的检查，情况就不一样了。一旦方法包含了这样的逻辑，单元测试就显得有意义了。

自动化集成测试

从使用的基本技术来说，自动化集成测试和单元测试在很多方面都很相似。你可以使用相同的测试运行器和构建系统的支持。自动化集成测试和单元测试的主要区别在于使用了相对较少的 mocking。

当一个单元测试简单地模拟从后端数据库返回的数据时，集成测试则会使用一个真实的数据库来测试。数据库是一个你需要的测试资源类型以及能暴露问题的极好例子。

自动化集成测试可能会很棘手，在选择时需要小心。

假如你在测试一个只读的中间件适配器，例如数据库的 SOAP 适配器，可能需要使用产品数据库的拷贝来做测试。数据库的内容需要可预测和可重复，否则很难去编写和运行测试。

这里的附加价值在于我们在使用产品数据的拷贝。它可能包含了从头开始创建测试数据时很难预测到的数据。这和人工测试的需求一样。使用自动化集成测试，你需要更多的自动化而不是人工测试。对于数据库来说，这并不是很复杂。自动化的数据库备份和恢复是众所周知的操作。

在自动化测试中使用 Docker

在构建自动化测试实验台时，使用 Docker 会非常方便。它在功能性的级别上添加了一些单元测试的特性。如果你的应用由集群中的几个服务器组件构成，你可以用一些容器来模拟整个集群。Docker 为集群提供了虚拟网络，在网络层面上让集群中的容器交互。

Docker 可以很容易将容器恢复到一个已知的状态。如果你在 Docker 容器中运行测试数据库，你可以很轻松地将数据库恢复到测试开发之前的相同状态。这和单元测试中 `After` 方法恢复环境类似。

持续集成服务器 Jenkins 已经支持启动和停止容器，在使用 Docker 做自动化测试时会比较有用。

使用 Docker Compose 来运行你需要的容器也是一个有用的选择。

Docker 还很年轻，使用 Docker 做测试自动化有时候需要编写并不优雅的胶水代码(glue code)。

举个简单的例子，启动可以交互的一个数据库容器和一个应用服务器容器。启动容器的基本过程比较简单，并且可以通过 shell 脚本或者 Docker Compose 来完成。但是，由于我们要在已经启动的应用服务器上运行测试，如何才能知道它已经正常启动了？在 WildFly 容器的场景下，除了监控输出日志中特定字符串的出现或者轮询 web socket 之外，没有什么明显方式判断容器的运行状态。在任何情况下，这些 hack 的方式都不甚优雅，而且实现起来比较耗时。尽管最终的结果还是能值回票价的。

Arquillian

Arquillian 是一个测试工具的例子，它的测试级别接近于集成测试而不是单元测试，同时可以 mock。Arquillian 特定于 Java 应用服务器，如 WildFly。Arquillian 很有趣，因为它说明了在测试过程中尽可能接近生产系统的努力。有很多种方式去达到这个目的，而实现的过程充满了权衡。

本书的源码包中有一个“hello world”风格的 Arquillian 的演示。

性能测试

性能测试是开发必不可少的一个部分，例如，对于大型的网站来说。

性能测试呈现了和集成测试类似的挑战。我们需要一个类生产环境的测试系统，从而使性能测试数据有助于预测真实系统的性能。

最常用的性能测试是负载测试。负载测试可以度量在性能测试软件产生综合请求时，服务器的响应时间。

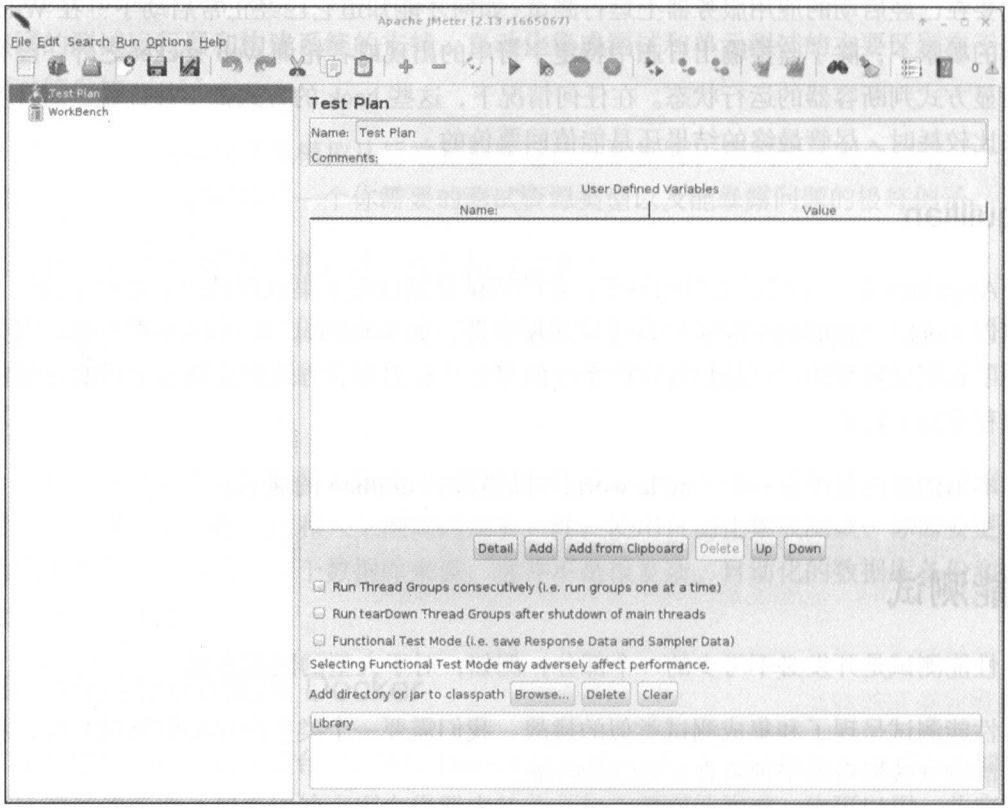
Apache JMeter 是一个开源的性能测试的工具。比起它的收费同行，如 LoadRunner，JMeter 更加简单，同时非常实用，简单并不是一件坏事。

JMeter 可以生成虚拟负载并测量许多协议的响应时间，例如 HT、LDAP、SOAP 和 JDBC。

有一个 JMeter 的 Maven 插件，可以在 JMeter 运行作为构建的一部分使用。

JMeter 也能用于持续集成服务器。Jenkins 有一个叫作 performance 的插件，可以执行 JMeter 的测试场景。

在理想情况下，持续集成服务器会部署代码到类产品的测试环境中。部署之后会执行性能测试，收集的测试数据如下面的截图所示：



自动化接受测试

自动化接受测试是从用户的角度出发来保证测试有效性的一种方法。

Cucumber 是一种测试用例以文本写成并关联测试代码的测试框架。这种方式被称为**行为驱动开发（behavior-driven development）**。Cucumber 原本是用 Ruby 实现的，但是现在已经移植到了多种不同的语言。

从 DevOps 的角度来看，Cucumber 的吸引力在于它尝试将不同的角色结合在一起。Cucumber 中 feature 的定义是用对话的形式实现的，而且不需要任何编程技巧也可以完成。测试运行所需要的实际数据会从描述中提取出来，用于测试。

这么做的意图很好，但是实现 Cucumber 测试的难度并不那么一目了然。尽管语言的行为规范基本上是自由文本，但它们仍然需要简化和形式化，否则编写匹配的代码和从描述中提取数据会变得困难。这使得编写规范对于最初编写它们的角色失去了吸引力。之后就会变成程序员去编写规范，但他们不喜欢冗繁，更倾向于编写普通的单元测试。

和很多事情一样，这里的精髓在于合作。只有开发人员和产品负责人一起合作努力编写规范，Cucumber 才能发挥作用。

现在，让我们来看看“hello world”风格的 Cucumber 的小例子。Cucumber 测试是以扩展名为 feature 的纯文本文件实现的，看上去像下面这样：

```
Feature: Addition
  I would like to add numbers with my pocket calculator

  Scenario: Integer numbers
    * I have entered 4 into the calculator
    * I press add
    * I have entered 2 into the calculator
    * I press equal
    * The result should be 6 on the screen
```

feature 的描述与实现语言无关。Cucumber 测试代码的描述是通过名为 **Gherkin** 的词表完成的。

如果你用的是 Java 8 lambda 版本的 Cucumber，测试的一个步骤看起来像会像下面这样：

```
Calculator calc;
public MyStepdefs() {
  Given("I have entered (\\d+) into the calculator", (Integer i) -> {
    System.out.format("Number entered: %n\n", i);
    calc.push(i);
  });
  When("I press (\\w+)", (String op) -> {
    System.out.format("operator entered: %n\n", op);
    calc.op(op);
  });
  Then("The result should be (\\d+)", (Integer i) -> {
    System.out.format("result : %n\n", i);
```

```
        assertThat(calc.result(),i);
    });
```

和以前一样，你可以在本书源码包中找到完整的代码。

这只是一个简单的例子，但它可以立刻能展现出 Cucumber 的长处和短处。feature 的描述可读性很好。但是你必须在测试代码中使用正则表达式来匹配字符串。即使只是微调了 feature 描述，你也需要去调整测试代码。

自动化 GUI 测试

自动化 GUI 测试有很多可取的特点，不过实现起来也有些困难。其中一个原因是在开发阶段，用户界面的改动会比较多，按钮和控制会在 GUI 界面中移动。

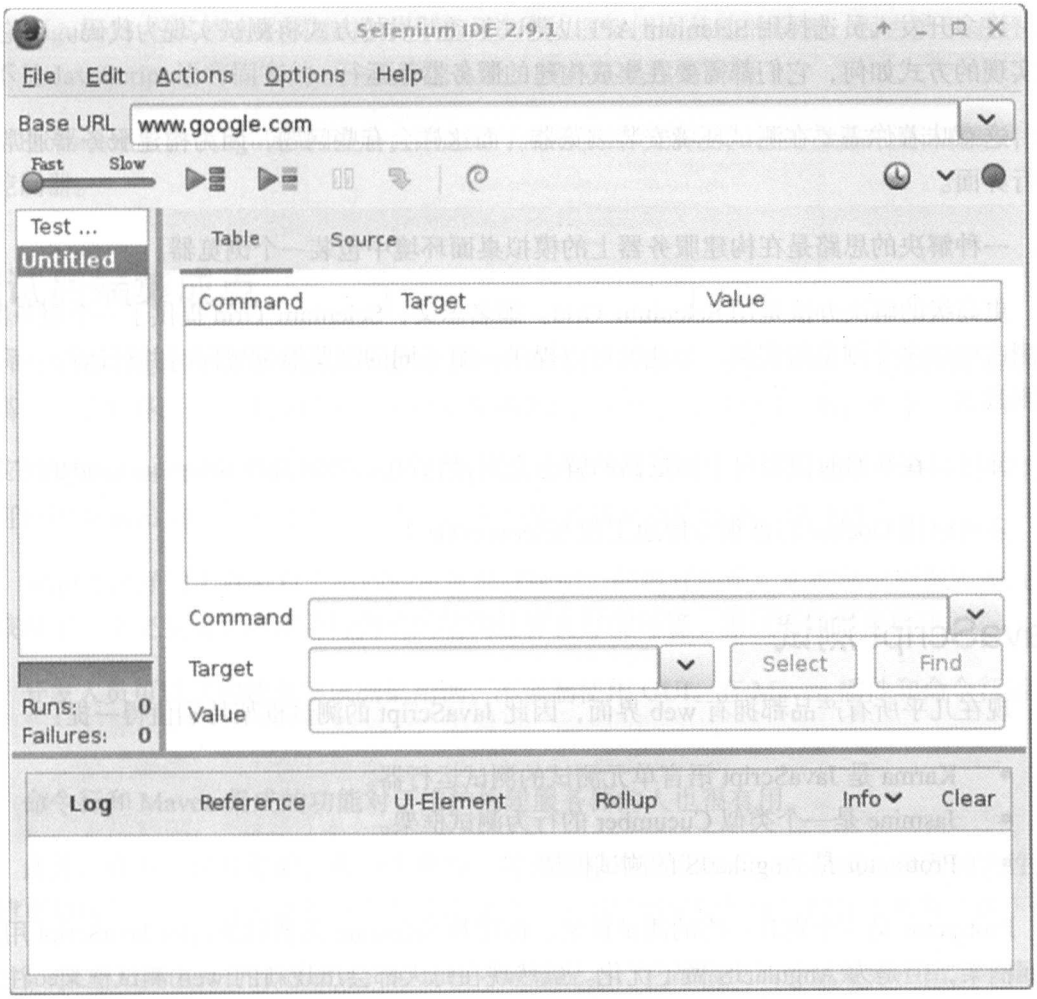
老一代的 GUI 测试工具是通过合成鼠标事件并将其发给 GUI 来工作的。当按钮移动时，模拟的鼠标单击事件点错了地方，测试失败。因此，根据 GUI 的改变来保持测试更新的成本变得很高。

Selenium 是使用了不同的、更加有效的方式的 web UI 测试工具包。控制器由标识符指引，因此 Selenium 可以通过检查文档对象模型（DOM）去找到控制器，而不是盲目地生成鼠标单击。

Selenium 在实际的使用中工作得很好，并且发展了很多年。

Sikuli 测试框架采取了另一种测试方法。它通过视觉框架 OpenCV 来帮助鉴别控制器，即便它们移动或者更改了外观。这对测试原生应用如游戏帮助很大。

下面的截图来自 Selenium 集成开发环境：



在 Jenkins 中集成 Selenium 测试

Selenium 通过激活浏览器并使其指向应用的 web 服务器，在自身集成到 JavaScript 和 DOM 层后，远程控制浏览器。

在测试实现时，有两种基本的办法：

- 记录浏览器中用户的交互行为，保存结果后可以让测试代码重用。
- 使用 Selenium 的测试 API 从头开始实现测试。

许多开发人员选择用 Selenium API 以测试驱动开发的方式将测试实现为代码。不论测试实现的方式如何，它们都需要在集成构建的服务器上运行。

这意味着你需要在测试环境安装浏览器。而这样会有些问题，因为构建服务器通常不运行界面。

一种解决的思路是在构建服务器上的模拟桌面环境中包装一个浏览器。

更高级的解决办法是用 Selenium Grid。顾名思义，Selenium Grid 提供了一个服务器，为测试生成多个浏览器实例。如此就可以提供一组不同的浏览器配置同时并行运行一系列的测试。

你可以在开始时使用单个浏览器的解决方案，然后在必要时选择 Selenium Grid 的方案。

还可以用 Docker 容器很方便地实现 Selenium Grid。

JavaScript 测试

现在几乎所有产品都拥有 web 界面，因此 JavaScript 的测试框架特别值得一提：

- Karma 是 JavaScript 语言单元测试的测试运行器。
- Jasmine 是一个类似 Cucumber 的行为测试框架。
- Protractor 是 AngularJS 的测试框架。

Protractor 是一个别具一格的测试框架，作用和 Selenium 类似的流行的 JavaScript 用户界面框架，但是为 AngularJS 做了优化。虽然现在每天都会出现新的 web 测试框架，注意到在可以使用 Selenium 测试来测试 AngularJS 应用的前提下，像 Protractor 这样的测试框架依旧存在，是一件很有趣的事情。

首先，Protractor 在底层使用了 Selenium web 驱动实现。

你可以用 JavaScript 来实现 Protractor 测试，如果不喜欢用 Java 实现测试，你也可以用 JavaScript 实现 Selenium 的测试用例。

使用 Protractor 的好处在于它内建了对 AngularJS 的支持，而像 Selenium 这样通用的框架不能做到这点。

AngularJS 有其特有的模型/视图设置。其他的框架使用其他的设置，因为模型/视图设置不是 JavaScript 语言固有的——不管怎么说，现在还是没有。

Protractor 了解 Angular 的特点，因此对于特殊的结构来说，它更容易在测试代码中定位控制器。

测试后端集成点

后端的功能性自动化测试，例如对 SOAP 和 REST 端点（endpoint），通常性价比都比较高。后端的界面通常比较稳定，所以对应的测试维护成本比起 GUI 测试要小一些。

用类似 soapUI 这样可以编写和执行测试的工具可以相对比较容易地实现测试。这些测试可以用 Maven 在命令行中运行，对于在构建服务器上做持续集成用处很大。

soapUI 这样的工具是对于不同的角色都有吸引力的绝好例子。构建测试用例的测试人员获得了一个交互式实现和运行测试的结构还算良好的环境。测试可以增量构建。

开发人员可以在构建中集成测试用例，不强制使用 GUI。有 Maven 插件和命令行运行器。

命令行和 Maven 集成的功能对于维护构建服务器的人也很有用。

此外，许可证是开源的，在一个单独、付费的版本中添加了一些功能。开源的天性让构建更加可靠。由于许可证意外到期或浮动许可证用尽而导致构建失败让人非常焦虑。

soapUI 工具也有自身的缺点，但是在一般情况下，它都很灵活并且效果不错。下面是它的用户界面的样子：

soapUI 的用户界面很直观。左边是树状视图的测试用例列表。可以选择单个测试或者整个测试套件运行。结果展示在右边的区域。

还有值得注意的是它的测试用例都是用 XML 定义的。这可以将它们作为代码在源码仓库中管理。也可以在需要的时用文本编辑器去修改它们，例如，当我们需要全局搜索并且替换一个改了名的标识码——这是我们 DevOps 喜欢的方式！

测试驱动开发

测试驱动开发（TDD）更侧重于测试自动化。20 世纪 90 年代的极限编程运动让它变得流行起来。

TDD 的通常被描述为如下的事件序列：

- **实现测试**：顾名思义，你开始编写测试，而后编写代码。一种实践的方式是先指定要开发代码的接口规格，然后实现代码。为了能编写测试，开发人员必须得找到所有相关的需求规格、用例和用户故事。

  将重心从编码切换到理解需求，对于正确的实现是很有裨益的。

- **验证新写的测试会失败**：新添加的测试会失败，因为还没有实现正确的行为，才刚编写测试需要存根和接口。运行测试并且确认它会失败。

- **编写实现测试的功能**：我们编写的代码不需要多么优雅或者高性能。开始时，只要让新测试通过即可。

- **验证新的测试和旧的测试会一起通过**：新的测试通过时，我们知道新实现的特性是正确的。旧的测试也通过，说明我们没有破坏已有的功能。

- **重构代码**：“重构”这个词来源于数学。在编程中，它的意思是清理代码的同时，让代码更容易理解和维护。我们需要重构是因为在前面的开发当中小小地耍了点诈。

TDD 是和 DevOps 相符合的风格，但它并不是唯一的一个。主要优势在于好的测试套件可以用在持续集成的测试中。

REPL（交互式命令行）驱动开发

REPL 驱动的开发并不是一个广泛认可的名词，它是我喜欢的开发风格，对测试有特定的影响。在使用解释型语言的时候很常见，如 Lisp、Python、Ruby 和 JavaScript 等。

在你使用读取、计算、打印、循环（REPL）类型语言时，你可以编写小而独立的函数，并且不依赖于全局的状态。

函数在编写的时候就得到了测试。

这种开发的方式和 TDD 有些区别。它侧重于编写没有或者有很少的副作用的函数。让代码更易于理解，而不是像 TDD 那样，在实现功能代码前编写测试用例。

你可以把这种开发方式和单元测试结合起来。因为你也可以用 REPL 驱动开发的方式去实现测试，这样的结合是一种很有效的策略。

一个完整的自动化测试场景

我们已经看过了很多不同的自动化测试是如何工作的。如何把每个部分组合成一个整体可能会很艰难。

在这一部分，我们会查看一个完整自动化测试的例子，继续使用我们企业的用户数据库 web 应用，Matangle。

你可以在本书对应的源码包中找到源代码。这个应用有以下几层：

- 一个 web 前端。
- 一个 JSON/REST 服务接口。
- 一个应用服务后端层。
- 一个数据库层。

测试代码在执行过程中会通过以下的步骤：

- 后端代码的单元测试。
- 使用 Selenium web 测试框架，针对 web 前端的功能性测试。
- 使用 soapUI，针对 JSON/REST 接口的功能性测试。

顺序执行全部测试，当它们都成功后，其结果可以作为判断应用栈是否可以部署到测试环境的标准，之后就可以进行人工测试了。

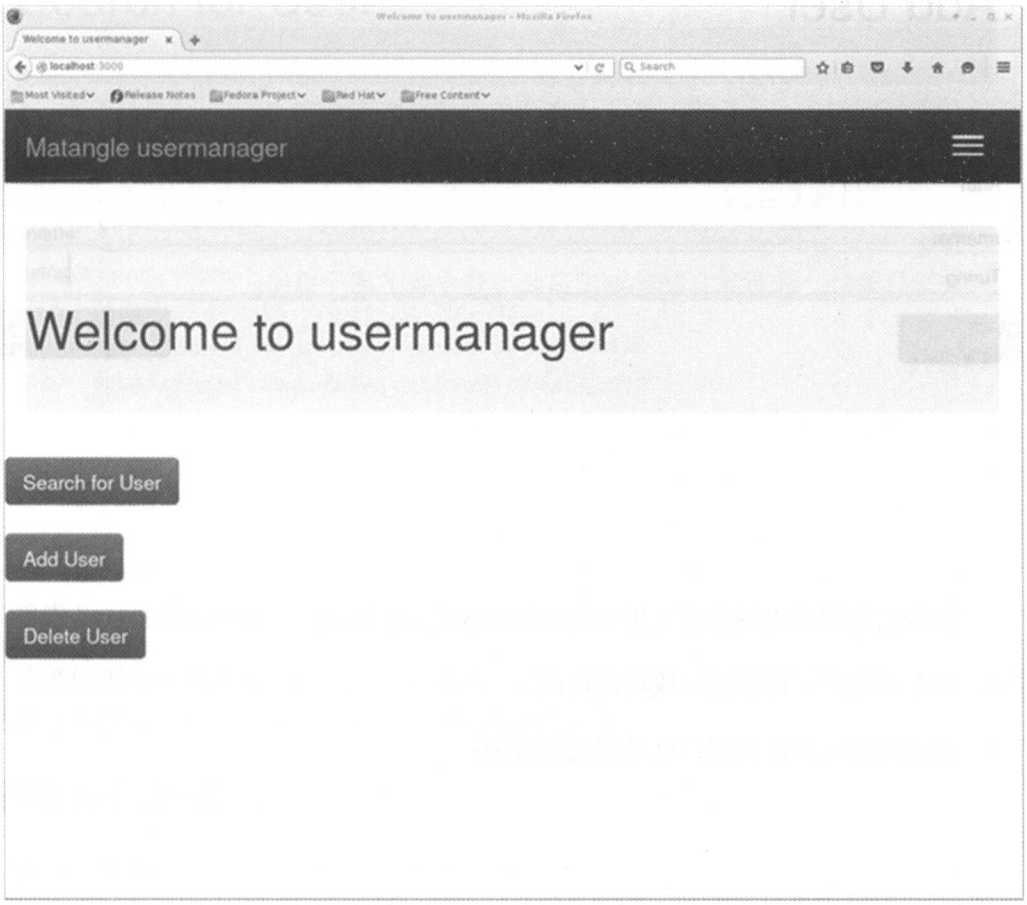
人工测试 web 应用

在我们自动化一些有用的东西前，我们需要理解被自动化的东西的细节。我们需要某种形式的测试计划。

以下是我们 web 应用的测试计划。它详细地介绍了在没有测试自动化的前提下，人工测试应该执行的步骤。这和真正的测试计划很相似，只是真正的计划通常会有更多的格式。在我们的例子中，会直接进入测试的细节：

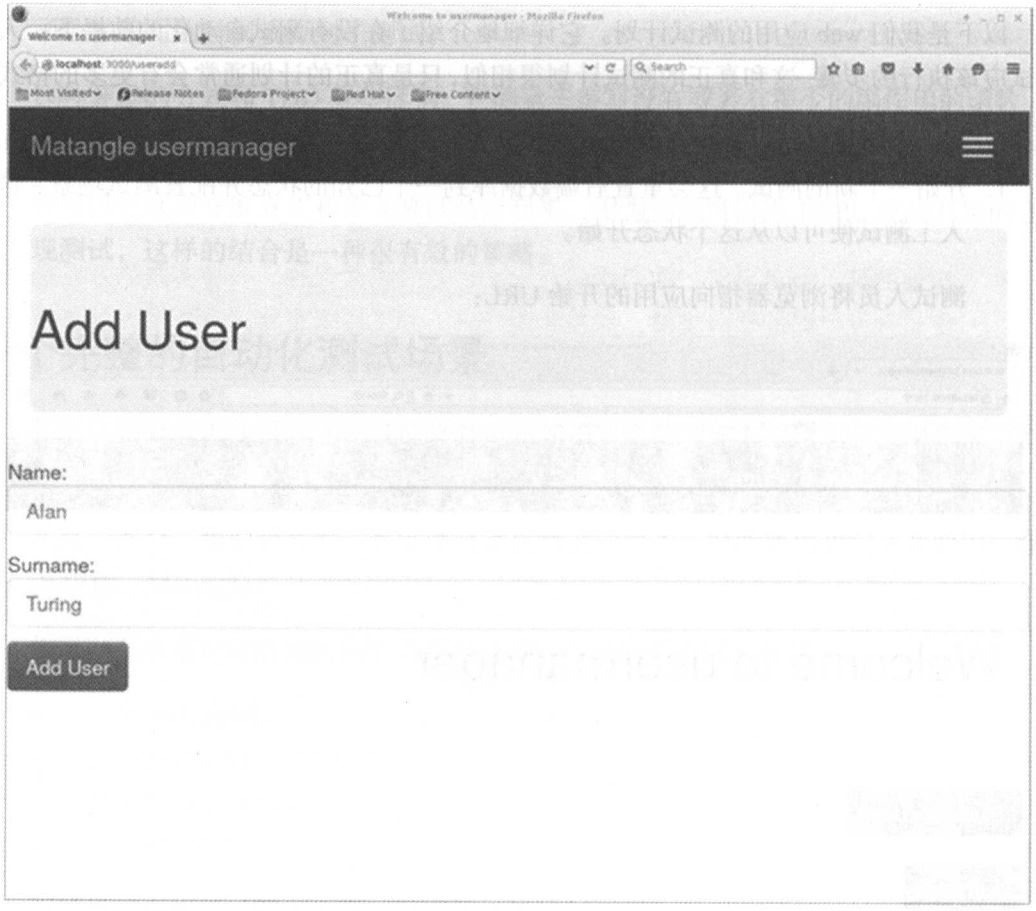
1. 开始一个新的测试。这会重置后端数据库到一个已知的状态并配置测试场景，这样人工测试便可以从这个状态开始。

   测试人员将浏览器指向应用的开始 URL：



2. 单击 **Add User**（增加用户）链接。

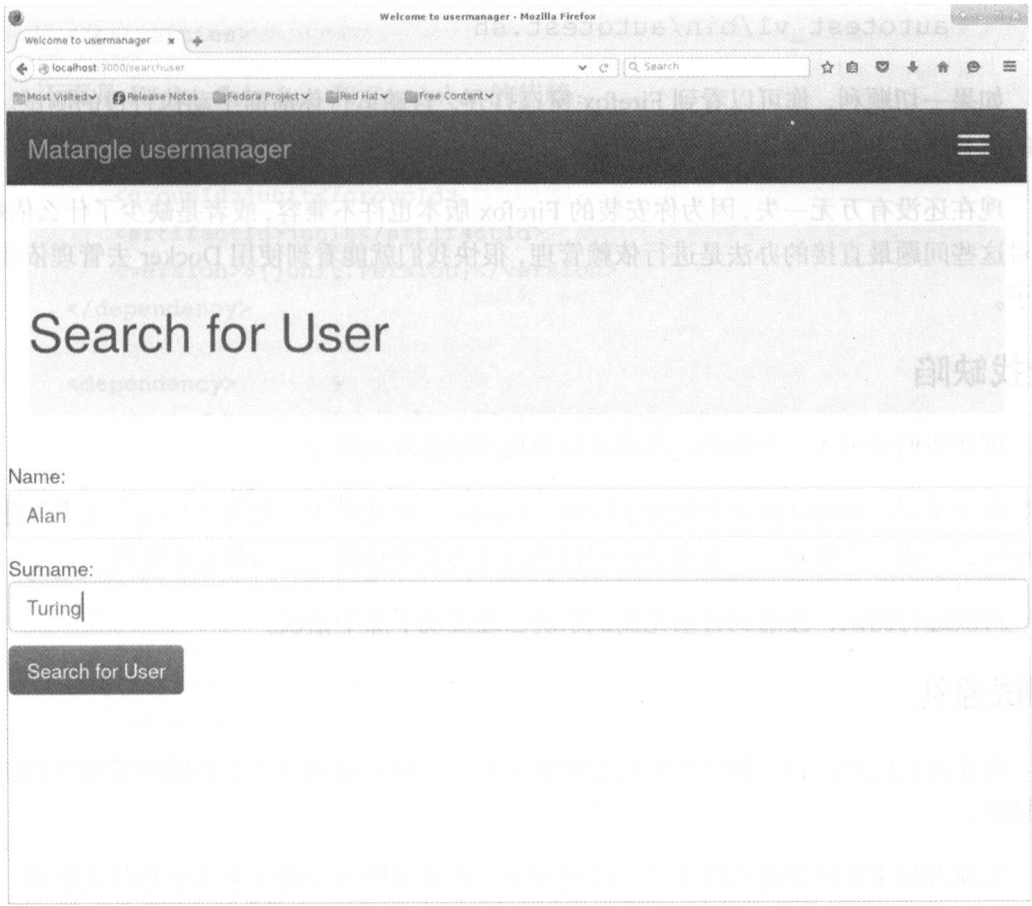
3. 添加一个用户：



在我们的测试用例中输入用户名——`Alan Turing`。

4. 保存新用户。将会显示成功的页面。

5. 通过搜索来验证该用户已经被成功添加：



单击 **Search for User** 链接。搜索 **Alan Turing**。确保 **Alan** 出现在结果列表中。

读者们现在可能不觉得应用的复杂度有多高，但如果要自动化这个场景，就需要关注细节到这个程度，我们在这里研究的只是这样的复杂性。

运行自动化测试

你可以在源码包中找到很多方式实现的测试。

要运行第一个，你需要安装 Firefox。

选择名为 `autotest_v1` 的测试，并在命令行中运行：

```
autotest_v1/bin/autotest.sh
```

如果一切顺利，你可以看到 Firefox 窗口打开，自动运行你前面手动执行过的测试。输入的值和手动单击的链接都会被自动化。

现在还没有万无一失，因为你安装的 Firefox 版本也许不兼容，或者是缺少了什么依赖。应对这些问题最直接的办法是进行依赖管理，很快我们就能看到使用 Docker 去管理依赖的例子。

查找缺陷

现在我们会引入一个缺陷，然后让自动化测试系统找到它。

作为练习，在测试资源中找到字符串“Turing”。修改其中一处为“Tring”或者其他的书写错误。改一个就好了，否则验证的代码会认为没有错误，一切都是正常的。

再次运行测试，注意到自动化测试系统已经发现了那个错误。

测试巡礼

现在我们已经运行了测试并确认它们能工作了。同时还确认了它们能够发现我们创建的缺陷。

实现看起来应该是什么样子呢？代码很多，在本书中再印刷出来不会有什么帮助。代码当然还是很有用的，浏览一下代码，查看一些具体的代码段。

打开 `autotest_v1/test/pom.xml` 文件。这是一个 Maven 项目的对象模型文件，它配置了所有测试使用的插件。Maven 的 POM 文件是声明式的 XML 文件，测试的步骤是一步一步地执行指令，所以在后一种情况下使用的是 Java。

顶部是属性块，里面会保存依赖的版本。这里并不需要修改版本号，放在这里是为了让 POM 文件的其他部分减少版本依赖：

```
<properties>
  <junit.version>XXX</junit.version>
  <selenium.version>XXX</selenium.version>
  <cucumber.version>XXX</cucumber.version>
  ...
```

```
</properties>
```

下面是 JUnit、Selenium 和 Cucumber 的依赖：

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
</dependency>

<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>${selenium.version}</version>
</dependency>

<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-core</artifactId>
    <version>${cucumber.version}</version>
</dependency>

<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>${cucumber.version}</version>
</dependency>

<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>${cucumber.version}</version>
</dependency>
```

要根据 Cucumber 的方法去定义测试，我们需要 feature 文件用人类可读的语言去描述测试步骤。这个 feature 文件对应了我们前面人工测试的测试计划：

```
Feature: Manage users
```

```
As an Administrator
I want to be able to
- Create a user
- Search for the user
- Delete the user

Scenario: Create a named user
   Given a user with the name 'Alan'
   And the surname 'Turing'
   When the administrator clicks 'Add User'
   Then the user should be added

Scenario: Search for a named user
   Given a user with the name 'Alan'
   And the surname 'Turing'
   When the administrator clicks 'Search for User'
   Then the user 'Alan Turing' should be shown

Scenario: Delete a named user
   Given a user with the name 'Alan'
   And the surname 'Turing'
   When the administrator clicks 'Delete User'
   Then the user should be deleted
```

feature 文件大部分是文本，还有一小部分机器可读的标记元素。由相应的测试代码使用正则表达式去解析场景的文本。

feature 文件也可以由你们团队使用的语言来本地化。这对于不习惯用英语的人来说很有帮助。

feature 需要真正的代码来执行，所以需要用某种方式去将 feature 绑定到代码。

你需要一个带有某些注解的测试类让 Cucumber 可以和 JUnit 一起工作：

```
@RunWith(Cucumber.class)
@Cucumber.Options(
       glue = "matangle.glue.manageUser",
       features = "features/manageUser.feature",
```

```
        format = {"pretty", "html:target/Cucumber"}
)
```

在这个例子中，按照约定，Cucumber 测试类名都有个 Step 的后缀。

现在你需要把测试方法和 feature 场景绑定在一起，并且从 feature 描述中解析出参数，传给测试方法。Java 的 Cucumber 版本基本上都是用注解实现的。这些注解对应于 feature 文件中使用的关键字：

```
@Given(".+a user with the name '(.+)'")
    public void addUser(String name) {
```

在这个用例中，不同的输入保存在成员变量中，直到整个用户界面事务准备完毕。操作的顺序取决于 feature 文件中实现的顺序。

为了证明 Cucumber 可以有不同的实现，本书的源码包中还有一个 Clojure 的例子。

到目前为止，我们已经明白，需要 Selenium 的一些类库和 Cucumber 去运行测试，Cucumber 的 feature 描述和我们的测试代码类绑定在一起。

下一步是检查 Cucumber 如何测试运行 Selenium 测试代码。

Cucumber 测试的步骤基本上就是调用后缀为 View 的类，类中包含了 Selenium 实现的细节。这在技术上不是必需的，但是可以让测试步骤的类更加易读，因为 Selenium 框架相关的内容都在一个单独的类中。

Selenium 框架负责测试代码和浏览器的交互。视图类是我们要自动化的 web 页面的抽象。视图代码中有 HTML 控制器对应的成员变量。你可以用 Selenium 框架的注解描述测试代码的成员变量和 HTML 元素之间的绑定关系，如下所示：

```
@FindBy(id = "name") private WebElement nameInput;
@FindBy(id = "surname") private WebElement surnameInput;
```

测试代码之后会使用成员变量去自动化与测试人员根据测试计划做出的相同步骤。将类划分为视图和步骤类也让步骤类和测试计划的相似性更加明显。当人们用代码参与到测试和质量保证的工作时，这样划分的考虑是有用处的。

要发送一个字符串，需要使用方法去模拟用户在键盘上输入：

```
nameInput.clear();
nameInput.sendKeys(value);
```

还有一些有用的方法，比如 `click()`，可以控制用户单击行为。

用 Docker 处理棘手的依赖

因为我们在测试代码例子中使用了 Maven，它处理了除浏览器之外所有的代码依赖。虽然你可以在兼容 Maven 的仓库中部署如 Firefox 的浏览器，以这种方式处理测试依赖，但这不是浏览器这个问题的一般处理方式。浏览器是很挑剔的，它们在不同的版本里行为不尽相同。我们需要一种机制来运行许多不同版本的各种浏览器。

幸运的是，有这样一个叫作 Selenium Grid 的机制。因为 Selenium 有可插拔驱动程序的架构，你可以很轻松地在一个客户端服务器架构中将浏览器后端分层。

要使用 Selenium Grid，你必须首先决定如何运行服务器的部分。最简单的方式就是使用在线提供商，出于说教的原因，我们先不在这里讨论这种方式。

在 `autotest_seleniumgrid` 目录中包含了使用 Docker 去启动本地 Selenium Grid 环境去运行测试的脚本。你可以通过例子中的脚本来运行测试。

关于如何运行 Selenium Grid 的最新信息可以在项目的 GitHub 页面上找到。

Selenium Grid 有分层的架构，设置它需要三个部分：

- 测试代码中有一个 `RemoteWebDriver` 实例。这将会是 Selenium Grid 的界面。
- 可以被看作浏览器实例代理的 Selenium Hub。
- Firefox 或者 Chrome 网格节点。这些都是被 Hub 代理的浏览器实例。

设置 `RemoteWebDriver` 的代码看起来像这样：

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setPlatform(Platform.LINUX);
capabilities.setBrowserName("Firefox");
      capabilities.setVersion("35");
   driver = new RemoteWebDriver(
            new URL("http://localhost:4444"),
            capabilities);
```

代码要求一个有特定组合功能的浏览器实例。系统会尽可能满足需求。

代码只有在 Selenium Grid Hub 有一个附加的 Firefox 节点时才能工作。

下面是如何启动用 Docker 打包的 Selenium Hub 的命令：

```
docker run -d -p 4444:4444 --name selenium-hub selenium/hub
```

以及如何启动一个 Firefox 节点并附加在 Hub 上的命令：

```
docker run -d --link selenium-hub:hub selenium/node-firefox
```

总结

这就是测试代码的全部。当阅读代码时，你可能只想使用本章阐明的部分方法。也许 Cucumber 并非真的适合你，或者你觉得例子中使用分层抽象的简明代码很有价值。这是很自然和合理的反应。采用适合你团队的方式。另外，在决定什么适合你时，看看源码包中其他可用的测试代码。

软件测试是一个蔚为大观的庞大主题。在本章中，我们调研了很多不同类型的可用测试。也看了在持续集成服务器上运行自动化软件测试的具体方法。我们还使用 Jenkins、Maven、JUnit 和 JMeter。虽然这些工具是面向 Java 的，其概念可以很容易转移到其他环境。

既然已经构建和测试了代码，我们将在下一章开始部署代码。

7

部署代码

代码的构建和测试现在已经完成了，接下来需要将其部署到服务器上，这样我们的客户就能使用新部署的特性了。

在部署这个领域有很多有竞争力的工具和选择，最适合你和你的企业的工具取决于具体的需求。

我们将探索 Puppet、Ansible、Salt、PalletOps 和其他的一些工具，并展示在不同的场景下部署示例应用。任一个工具都有其对应的补充服务和工具的生态系统，学习哪一个都不是一件简单的事情。

在本书中，我们碰到过已经存在的不同的部署系统的各个方面。我们见过 RPM 和 `.deb` 文件，以及如何用 `fpm` 命令构建它们。我们也看过不同的 Java 工件以及 Maven 如何使用二进制数据仓库来部署不同版本的工件。

在本章，我们重点关注安装二进制数据包以及用配置管理系统安装它们的配置。

为什么有这么多的部署系统

在真实的服务器上安装和配置包的选项丰富到让人迷惑，更不用说所有的部署客户端代码的方式。

让我们先来检查一下要解决的基本问题。

现在有个典型的企业级应用，包含了很多不同的高级组件。为了开始探讨在这个领域

内存在的挑战，我们不需要把场景设置得过于复杂。

在我们的应用场景下，我们有：

- 一个 web 服务器。
- 一个应用服务器。
- 一个数据库服务器。

如果我们只有一个物理服务器和这些一年左右才发布的少量组件，可以手动安装软件完成任务。这是处理这种情况最经济的方式，即使手动完成很烦人并且容易出错。

期望在现实中存在这么简单的发布周期并不合理。对于一个拥有超过上百台服务器和署应用的大型企业来说，更加可能的情况是它们的部署需求和部署本身都是不同的。

管理现实中展示出来的复杂性是很困难的，所以有很多不同的方式解决相同问题的事情就不难理解了。

不论执行我们代码的基本单元是什么，是实体物理机、虚拟机、某种形式的容器技术或是以上的综合体，都有一些挑战需要我们解决。现在让我们来看看。

配置基础操作系统

不管如何，基础操作系统的配置必须要处理。

通常情况下，我们的应用程序栈对基础操作系统有微妙或者不那么微妙的依赖。有些应用程序技术栈，比如 Java、Python 或者 Ruby，对操作系统的依赖不那么明显，因为这些技术都提供了跨平台支持的功能。在其他情况下，对操作系统的依赖是显而易见的，比如当你使用底层混合的硬件和软件集成时，这在电信行业里面很常见。

处理这种基本问题有很多现成的解决方案。有些系统使用裸机（或裸虚拟机）的工作方式，它们需要从头开始安装操作系统，然后再安装企业需要的服务器的所有依赖。这样的系统包括例如 Red Hat Satellite 和 Cobbler，它们工作的原理相似，但是 Cobbler 更加轻量。

Cobbler 允许你通过使用 `dhcpd` 的网络来引导启动物理机或者虚拟机。然后 DHCP 服务器可以提供给你一个兼容网络启动的镜像。当网络引导的镜像启动时，它联系 Cobbler 去获得为了创建新的操作系统所依赖的所有包。服务器从目标机器上，通过如网络 MAC 地址来决定安装哪些包。

另外一种现在流行的方式是提供可复用的基本操作系统镜像。像 AWS、Azure 或者 OpenStack 等云系统就这样工作的。像 Docker 这样的容器系统也是这样工作的，首先声明使用的基础容器镜像，然后说明要定制的镜像的修改。

描述集群

一定有一种方式可以描述集群。

如果你的企业只有一台服务器，运行一个应用，那你可能不需要描述如何将应用部署在集群上。不幸的是（或者幸运的是，根据你的眼界），现实的情况通常是你的应用在一组机器、虚拟机或物理机上运行。

本章用到的所有的系统都以不同的方式支持集群。Puppet 有一个扩展的系统允许机器拥有不同的角色，这些角色反过来代表一组包和配置。Ansible 和 Salt 也有这类系统。基于容器的 Docker 系统有一个新兴的基础设施，可以描述连接在一起的容器以及 Docker 能接受和部署这样的集群描述的 Docker 主机集群。

像 AWS 这样的云系统也有对应的方式和描述符去实现集群部署。

集群描述符通常也用来描述应用层。

为系统交付包

一定有一种方式可以为系统交付包。

很多应用都可以通过配置管理系统用包的形式在目标系统上不加修改地安装。像 RPM 和 deb 这样的包系统有很多优点，比如通过为包中所有文件提供校验码，可以验证包提供的文件没有在目标系统上被修改。从安全和调试的角度来说很有用处。包的交付通常是通过操作系统的工具如 RedHat 上的 yum 包管理系统，但有时候配置管理系统也可以自己用自己的工具交付包和文件。这些工具通常和操作系统的包管理工具串联使用。

必须有独立于安装的包之外的配置管理的方式。

显而易见，配置管理系统需要能管理我们应用的配置。不管曾经付出了多少努力去统一这一领域，因为应用间配置方法的不同，导致了配置管理的复杂性。

配置应用程序最常见和最灵活的系统都使用基于文本的配置文件。还有些其他方法，

如提供 API 去处理配置（如命令行接口）的应用或者通过数据库设置去处理配置。

根据我的经验，基于文本文件的配置管理系统带来的麻烦最少，至少应该优先为内部代码使用。有很多种方式去管理基于文本的配置。你可以使用源码处理系统管理，如 Git。通过很多工具，如 `diff`，可以降低调试出错配置的难度。如果情况紧急，你可以通过远程文本编辑器，如 Emacs 或者 Vi，在服务器上直接编辑配置。

通过数据库来处理配置的方式不那么灵活。这是一种容易引起争议的反模式，通常在开发和运维团队隔得比较远的企业中出现，也是 DevOps 希望解决的目标。用数据库处理配置让应用栈运行更加困难。你需要一个工作的数据库才能让应用启动。

通过命令式的命令行 API 来管理配置设置同样也是让人怀疑的实践，但是有时候还是有用的，特别是在 API 用来管理底层基于文本的配置时。很多配置管理系统，如 Puppet，依赖于能够管理声明配置。如果我们通过其他方式管理配置，比如命令行命令式 API，就失去了使用 Puppet 的好处。

甚至管理基于文本的配置也会带来问题。对于应用来说，它们可以发明属于自己的配置文件格式，但是有一组基本的文件格式比较受欢迎。比如 XML、YML、JSON 和 INI 的文件格式。

配置文件通常不是静态的，因为如果它们是静态文件，那么你就可以使用包系统把它们打包成二级制工件去部署。

一般来说，应用配置文件需要基于一些模板文件，之后再被实例化为适合于将要部署应用的机器的格式。

比如应用的数据库连接描述符。如果你在测试环境部署应用，你想让连接描述符指向测试环境服务器。同理，如果在产品服务器上部署，你希望你的连接指向的是产品环境的数据库服务器。

顺便说一句，有些企业试图通过管理自己的 DNS 服务器来解决这个问题，比如样例数据库 DNS 别名为 `database.yourorg.com`，在不同的环境中被解析为不同的服务器。当然 `yourorg.com` 这个域名需要根据你们企业的情况来替换，数据库服务器也是一样。

根据不同的环境使用不同的 DNS 解析器是一个很有用的策略。然而对于开发人员来说，在自己的开发环境上使用相同的策略会比较困难。在开发环境的机器上运行私有的 DNS 服

务器比较麻烦，管理本地的主机文件也很烦琐。在这些场景下，简单点的方法就是将数据库主机和其他应用层级的后端系统作为可以配置的选项。

很多时候，我们可以完全忽略实际的配置文件格式的细节，只是依靠配置系统的模板管理系统。通过为占位符提供特殊语法的方式是可行的：当应用在将要部署的具体服务器上创建配置文件时，由配置管理系统替换占位符。你可以用相同的方式来处理所有基于文本的配置文件，甚至偶尔处理一下二进制文件，即使这样的方式需要尽量避免使用。

XML 格式的工具和基础设施在配置管理中颇为有用，同时 XML 也是一种流行的配置文件格式。例如，有种特殊的语言 XSLT，可以将 XML 从一种结构转换为另一种结构。在某些场景下非常有用，但在具体的实践中并没有想象中使用得那么频繁。

简单的模板宏替换的方法会给你带来意想不到的好处，并适用于几乎所有基于文本的配置格式。XML 也相当冗长，这使得它在某些圈子中不受欢迎。YML 可以视作 XML 冗长的对立，少打一些字，但是可以完成和 XML 一样的事情。

对于一些文本配置系统，另一个值得一提的实用特性是一个基础配置文件可以引用其他配置文件。一个例子就是标准的 Unix sudo 工具，它在`/etc/sudoers` 文件中有自己的基础配置，同时允许通过引用安装在`/etc/sudoers.d` 目录下的其他文件做本地定制。

这样非常方便，因为你可以提供一个新的 `sudoer` 文件而不用担心已有的配置。这允许更大程度的模块化，在应用程序支持时是一种很方便的模式。

虚拟化栈

有自己服务器集群的企业更倾向于使用虚拟化，以便封装他们应用的不同组件。

根据你的需求，有很多相应的不同解决方案。

虚拟化解决方案提供了虚拟机，具有如网络设备和 CPU 的虚拟硬件。有时候我们会混淆虚拟化和容器技术，因为它们有一些相似之处。

你可以使用虚拟化技术去模拟和物理硬件完全不同的硬件。这通常被称为仿真。如果你想在自己的开发机器上仿真移动设备去测试移动应用，你可以使用虚拟化技术去仿真设备。目标平台越接近底层硬件，模拟器在仿真的过程中效率越高。举个例子，你可以用 QEMU

模拟器仿真一个 Android 设备。如果在 x86_64 位的开发机器上仿真一个 Android x86_64 设备，要比在 x86_64 位的开发机器上仿真基于 ARM 的 Android 设备效率更高。

使用服务器虚拟化技术，你通常不用担心仿真的可能性。相反，你更在意封装应用的服务组件。例如，如果一个应用服务器组件疯狂运行并且不合理地消耗大量的 CPU 时间以及其他资源，你不会希望整个物理机完全停止服务。

通过在一个 64 核的机器上创建一个双核的虚拟机，也许可以解决这个问题。在应用运行时 CPU 只有两个核心受到影响。同理内存的分配也是一样。

基于容器的技术提供了和虚拟化技术类似的封装及资源分配控制。虽然容器通常不提供虚拟化的仿真特性。这并不是问题，因为我们很少需要模拟应用服务器。

抽象底层硬件和在相互竞争的不同虚拟机之间调度硬件资源的组件称为**虚拟机监控程序（hypervisor）**。它可以直接运行在硬件上，在这种情况下也被称为裸机管理程序。除此之外，它也可以在操作系统内核的帮助下运行在操作系统中。

VMware 是一个专有的虚拟化解决方案，同时支持桌面和服务器虚拟机监控。很多企业都支持和使用它。有时候服务器虚拟机监控有不同的名字，现在它叫作 VMware ESX，是一个裸机监控程序。

KVM 是一个 Linux 下的虚拟化解决方案。它在 Linux 操作系统的主机上运行。它是开源的解决方案，通常比专有的解决方案更便宜，因为没有实例的授权费用，因此在大量采用虚拟化技术的企业里很流行。

Xen 是另外一种类型的虚拟化技术，在诸多特性中，它支持**半虚拟化（paravirtualization）**。半虚拟化来自客户操作系统可以使用修改的内核的思想，执行的效率更高。这种方式处于使用独立内核版本的完全的 CPU 模拟与使用宿主内核的基于容器的虚拟化之间。

VirtualBox 是来自于 Oracle 的开源虚拟化解决方案。在开发人员中很流行，有时候也会在服务器上安装但是体量不大。使用 Microsoft Windows 机器的开发人员通常会使用 VirtualBox 在本地来模拟 Linux 服务器环境。同样地，使用 Linux 作为其工作站的开发人员也会用 VirtualBox 去模拟 Windows 服务器。

不同类型的虚拟化技术的共同点在于它们提供了可以自动化虚拟机管理的 API。

libvirt 就是这样的 API，它可以用在几种不同的虚拟管理程序（hypervisor）底层，比如 KVM、QEMU、Xen 和 LXC。

在客户端执行代码

这里介绍的几种配置管理系统允许你复用节点描述符，在匹配的节点上执行代码。有时候这样很方便。例如，为了调试，你也许会想要在所有面向 Internet 的公网 HTTP 服务器上运行列出目录的命令。

在 Puppet 的生态系统中，这个命令执行系统被称为 Marionette Collective，简称为 MCollective。

有关练习的注意事项

尝试使用 Docker 通过不同的部署系统来管理我们将要实验的基础操作系统很容易。在特定的部署系统上开发和调试部署代码时，这种方式比较节省时间。这些代码之后可以用于部署物理机或虚拟机。

首先我们会在本地开发方式上尝试每个不同的部署系统。之后，通过将几个容器组成一个虚拟集群去模拟完整的部署。

我们尽量使用 Docker 官方的镜像，但是有时候会出现找不到或者官方镜像消失的情况，像 Ansible 的官方镜像一样。这就是 DevOps 快速向前的生活，或好或坏。

需要注意的是，在模拟一个完整的操作系统时 Docker 有一些限制。有时候容器需要在高权限的模式下运行。我们将会在碰到这个问题时处理。

还有就是很多人会选择 Vagrant 去做这些测试。如果可能我还是推荐使用 Docker，因为它更加轻量、快速，在很多情况下都足够使用。

请记住，在生产中实际部署的系统将需要更多关注于安全以及除上文的介绍以外的更多细节。

Puppet 服务器和 Puppet 代理

Puppet 是一个在大型企业中很流行的部署解决方案，也是最早的部署系统之一。

Puppet 是由一个客户端和服务器组成的解决方案，客户端节点定期检查 Puppet 服务器中本地的配置是否需要更新。

Puppet 服务器被称为**木偶大师（Puppet master）**，Puppet 很多组件都是以类似的文字游戏方式命名的。

Puppet 在处理服务器集群的复杂性上提供了很大的灵活性，因此，该工具本身也很复杂。

下面是一个 Puppet 客户端和 Puppet 服务器对话的一个示例场景：

1. Puppet 客户端决定是时候检查 Puppet 服务器上的配置是否有更改。可以通过计时器或者在客户端手动操作。客户端和服务器之间的对话通常是通过 SSL 加密的。
2. Puppet 客户端提供了它的凭证，这样 Puppet 服务器可以知道哪个客户端在请求。管理客户端凭证是一个单独的问题。
3. Puppet 服务器通过编译 Puppet 目录发现客户端需要的配置并将其发送给客户端。这涉及了一些机制，一个特定的设置并不需要利用所有的可能性。
   对于 Puppet 客户端来说，同时使用基于角色和具体配置的方式很常见。基于角色的配置可以被继承。
4. Puppet 服务器在客户端运行必要的代码，这样可以让 Puppet 服务器决定哪些配置匹配这个客户端。

从这点来看，Puppet 配置是表意的。你声明一个机器应该有什么样的配置，Puppet 帮你将客户端当前状态变为期望的状态。

Puppet 生态系统既有优点也有缺点：

- Puppet 的社区庞大，网络上也有很多相关的资源。Puppet 有很多模块，如果你的部署模块不是那么特别，那么应该已经有现成的类似模块，可以在其基础上稍加修改来满足你的需求。

- 在 Puppet 客户端机器上需要一系列 Puppet 的依赖。有时候这会带来问题。有时候 Puppet 代理依赖的 Ruby 运行时的版本比你操作系统发型版软件仓库中的版本更新。企业级的操作系统版本通常比较滞后。
- Puppet 配置实现起来可能会很复杂，测试也比较困难。

Ansible

Ansible 是一个简洁的部署解决方案。

Ansible 的架构中没有代理，它不需要像 Puppet 一样在客户端运行一个后台程序。相反，Ansible 服务器登录到 Ansible 节点，并且通过 SSH 执行命令去安装所需的配置。

虽然 Ansible 的无代理架构确实让事情变得简单，Ansible 的节点上还是需要有 Python 解释器的。相比 Puppet 在运行代码时依赖的 Ruby 版本，Ansible 对于代码运行时依赖的 Python 版本更加宽容些，所以对 Python 的依赖在实践中没有带来很大的麻烦。

和 Puppet 以及其他的部署系统类似，Ansible 也关注于幂等的配置描述符。这意味着描述符是表意的，并且 Ansible 系统知道如何将服务器变为期望的状态。你可以反复运行，这很安全，不过对于一个比较紧急的系统来说没有必要。

让我们用前面讨论过的 Docker 的方式来尝试 Ansible。

我们将使用为了这个目的开发的 `williamyeh/ansible` 镜像，但是我们也可以使用任意的 Ansible Docker 镜像或者其他的镜像，我们可以在后面添加 Ansible。

1. 用下面的语句创建一个 Dockerfile：

```
FROM williamyeh/ansible:centos7
```

2. 用下面的命令构建 Docker 容器：

```
docker build .
```

这样会下载镜像并且创建一个我们可以使用的 Docker 空容器。

通常来说，你会有一个更加复杂的 Dockerfile 去添加需要的东西，但是在这个场景下，我们希望以交互的方式使用镜像。所以我们会把宿主的 Ansible 文件目录映射

到容器中，这样我们可以在宿主机上修改文件，并且很容易反复运行。

3. 运行容器。

以下命令可以用来运行容器。你需要前面 build 命令生成的哈希码：

```
docker run -v `pwd`/ansible:/ansible -it <hash> bash
```

现在我们有了一个提示命令行，同时也可以使用 Ansible。-v 选项用来将宿主机的部分文件系统和 Docker 的客户容器共享。文件会在容器的/ansible 目录下显示。

playbook.yml 文件如下所示：

```
---
- hosts: localhost
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
```

这个 playbook 做的事情不多，但是它展示了 Ansible playbooks 的一些概念。

现在尝试运行我们的 Ansible playbook：

```
cd /ansible
ansible-playbook -i inventory playbook.yml    --connection=local --sudo
```

输出如下：

```
PLAY [localhost] ***************************************************************
******

GATHERING FACTS ****************************************************************
******
ok: [localhost]

TASK: [ensure apache is at the latest version] *************************
```

```
******
ok: [localhost]

PLAY RECAP *********************************************************************
******
localhost                  : ok=2    changed=0    unreachable=0
failed=0
```

任务顺序运行来确保我们期望的最终状态。在这个场景下，我们希望用 yum 安装 Apache 的 httpd，并且是最新版本的 httpd。

为了继续探索，我们可以添加更多的东西，比如自动启动服务。但是，这里我们会碰到使用 Docker 去模拟物理机或虚拟机的限制。Docker 毕竟是一种容器技术，不是全面的虚拟化系统。在 Docker 的常见使用场景下，这点并不重要，但是在我们的使用场景下，需要做一些变通才能继续。主要的问题是 systemd 初始化系统在容器中运行需要特殊照顾。Red Hat 的开发人员想出了解决的办法。下面是一个少许修改的 Docker 镜像版本，来自于在 Red Hat 工作的 Vaclav Pavlin：

```
FROM fedora
RUN yum -y update; yum clean all
RUN yum install  ansible sudo
RUN systemctl mask systemd-remount-fs.service dev-hugepages.mount sys-
fs-fuse-connections.mount systemd-logind.service getty.target console-
getty.service
RUN cp /usr/lib/systemd/system/dbus.service /etc/systemd/system/; sed
-i 's/OOMScoreAdjust=-900//' /etc/systemd/system/dbus.service

VOLUME ["/sys/fs/cgroup", "/run", "/tmp"]
ENV container=docker

CMD ["/usr/sbin/init"]
```

container 这个环境变量用来告诉 systemd 初始化系统，它在容器中运行，需要有对应的行为。

我们需要给 docker 传入更多的参数，以便让 systemd 在容器中运行：

```
docker run -it --rm -v /sys/fs/cgroup:/sys/fs/cgroup:ro  -v `pwd`/
```

```
ansible:/ansible <hash>
```

容器用 systemd 启动，现在我们需要从一个不同的 shell 命令行连接运行着的容器：

```
docker exec -it <hash> bash
```

我的妈呀，为了让容器运行得更加逼真我们也真是耗尽心力。不过换句话说，用虚拟机的方式，如 VirtualBox，在我看来更加麻烦。当然读者也许不会这么觉得。现在，我们可以在容器中运行一个更加复杂的 Ansible playbook，如下所示：

```
---
- hosts: localhost
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

这个例子在前面的基础上构建，并且告诉你如何：

- 安装包。
- 编写一个模板文件。
- 处理一个服务的运行状态。

格式为非常简单的 YML 语法。

PalletOps

PalletOps 是一个高级的部署系统，它结合了 Lisp 表意的能力以及一个轻量级的配置服务器。

PalletOps 采取了 Ansible 无代理的思路并且更进一步。你只需要在节点安装 `ssh` 和 `bash`，而不需要安装配置 Ruby 或者 Python 解释器。这些都是非常简单的要求。PalletOps 将 Lisp 定义的 DSL 代码编译成可以在 slave 节点上执行的 Bash 代码。这些需求如此简单，你可以在非常小和简单的服务器甚至是手机上运行。

从另一个角度讲，虽然也有一些 Pallet 支持的学名为篓（crate）的模块，但是相比 Puppet 或者 Ansible 来说就少了很多。

用 Chef 做部署

Chef 是 Opscode 开发的基于 Ruby 的部署系统。

使用 Chef 非常容易，为了好玩，我们可以在 Docker 容器中运行，这样我们的实验不会污染宿主机的环境：

```
docker run -it ubuntu
```

我们需要 `curl` 命令来下载 chef 的安装器：

```
apt-get -y install curl
curl -L https://www.opscode.com/chef/install.sh | bash
```

Chef 的安装器是用 Chef 团队开发的一个名为 omnibus 的工具开发而成的。在这里我们的目的是尝试叫作 `chef-solo` 的 Chef 工具。用下面的方式验证工具安装完成：

```
chef-solo -v
```

输出如下：

```
Chef: 12.5.1
```

这里使用 `chef-solo` 的原因是为了能在脱离配置管理系统基础架构的前提下运行配置脚本，比如客户端/服务器设置。这样的测试环境在使用配置管理系统时通常比较有用，

因为在开发即将部署的配置时，很难让所有的东西有序工作。

Chef 有自己推荐的文件结构，可以从 Github 上下载到一个预置的结构。可以用下面的命令下载和提取：

```
curl -L  http://github.com/opscode/chef-repo/tarball/master -o master.tgz
tar -zxf master.tgz
mv chef-chef-repo* chef-repo
rm master.tgz
```

现在你就有了一个适用于 Chef bookbook 的文件结构，像下面这样：

```
./cookbooks
./cookbooks/README.md
./data_bags
./data_bags/README.md
./environments
./environments/README.md
./README.md
./LICENSE
./roles
./roles/README.md
./chefignore
```

为了让一切都顺利工作，你还需要进一步告诉 chef 在什么地方能找到它的 cookbook：

```
mkdir .chef
echo "cookbook_path [ '/root/chef-repo/cookbooks' ]" > .chef/knife.rb
```

现在我们可以用 knife 工具为配置创建一个模板，如下所示：

```
knife cookbook create phpapp
```

用 SaltStack 做部署

SaltStack 是一个基于 Python 的部署解决方案。

Jackson Cage 制作了一个 docker 化的 Salt 测试环境。可以用以下命令启动：

```
docker run -i -t --name=saltdocker_master_1 -h master -p 4505 -p 4506 \
  -p 8080 -p 8081 -e SALT_NAME=master -e SALT_USE=master \
  -v `pwd`/srv/salt:/srv/salt:rw jacksoncage/salt
```

这样会创建一个包含 Salt master 和 Salt minion 的容器。为了进一步探索我们可以在容器内创建一个 shell 命令行：

```
docker exec -i -t saltdocker_master_1 bash
```

我们需要在服务器上应用一个配置。Salt 称配置为“state”或者 Salt states。

在我们的场景下，我们希望用简单的 Salt state 来安装一个 Apache 服务器：

```
top.sls:
base:
  '*':
    - webserver
webserver.sls:
apache2:            # ID declaration
  pkg:              # state declaration
    - installed     # function declaration
```

Salt 的配置文件使用 `.yml` 文件，和 Ansible 类似。

文件 `top.sls` 声明所有匹配的节点都是 `webserver` 类型。`webserver` 的 state 声明应当安装一个 `apapche2` 包，基本上就是这样。请注意这与操作系统发行版相关。我们使用的 Salt Docker 测试镜像基于 Ubuntu，在上面安装的 Apache 服务器的包名为 `apache2`。在 Fedora 上面的 Apache 服务器的包名为 `httpd`。

运行以下命令可以看到 Salt 的工作方式，读取 Salt state，然后在本地应用：

```
salt-call --local state.highstate -l debug
```

第一次运行的冗余信息比较多，尤其是我们添加了 `debug` 参数。

现在让我们再运行一次命令：

```
salt-call --local state.highstate -l debug
```

这也有很多冗余信息，输出的最后部分如下：

```
local:
----------
          ID: apache2
    Function: pkg.installed
      Result: True
     Comment: Package apache2 is already installed.
     Started: 22:55:36.937634
Duration: 2267.167 ms
Changes:

Summary
------------
Succeeded: 1
Failed:    0
------------
Total states run:     1
```

现在你可以退出并重启容器。这样可以清理上一次运行时安装 Apache 实例的容器。

这次我们会应用相同的 state，但是使用消息队列的方式而不是在本地应用 state：

```
salt-call state.highstate
```

和上次运行的命令一样，只不过我们去掉了`-local` 标识。你也可以尝试再次运行命令来检查 state 是否保持不变。

从执行的模型来比较 Salt、Ansible、Puppet 和 PalletOps

本章里我们使用的配置管理系统有很多的相似之处，在客户端节点运行代码的方式有很多的不同：

- 对于 Puppet 来说，一个 Puppet 代理在 Puppet 服务器上注册同时打开一个通信通道来获取命令。这个过程通常会定期运行，一般每半个小时一次。

30 分钟不算快。你也可以配置一个更短的运行时间间隔。不管以什么频率运行，Puppet 主要使用拉模型。客户端必须检查之后才知道那些修改可以用了。

- Ansible 通过 SSH 来推送期望的修改。这是推模式。
- Salt 使用的是推模式，但是实现的方式不同。它利用了 ZeroMQ 消息服务器，客户端连接到消息服务器，监听修改的通知。工作原理和 Puppet 有点像，但是速度更快。

哪种方式最好一直是开发者社区争执的一个话题。消息队列架构的支持者认为速度很重要，而且它的速度更快。SSH 方法的支持者认为简单更加重要而且它足够快。我个人倾向于后面的观点。事情总会出篓子，复杂度增加时出问题的概率就更大。

Vagrant

Vagrant 是一个虚拟机的配置管理系统。它面向开发人员来创建虚拟机，但是也可以用于其他目的。

Vagrant 支持几种虚拟化提供商，VirtualBox 是一个在开发人员中比较流行的提供商。

首先是一些准备工作。根据操作系统发行版的指南去安装 `vagrant`。在 Fedora 上运行的命令如下：

```
yum install 'vagrant*'
```

这会安装一系列的包。不过，在 Fedora 上安装时我们会碰到一些问题。Fedora 的 Vagrant 包使用 `libvirt` 作为虚拟机提供商而不是 VirtualBox。在很多场景下这已经足够用了，但是在我们的场景下更希望使用 VirtualBox 作为提供商，这就需要在 Fedora 上做一些额外的工作了。如果你用的是其他的操作系统发行版，情况可能会不太一样。

首先，给你的 Fedora 系统添加 VirtualBox 的仓库源。然后用以下的 `dnf` 命令安装 VirtualBox：

```
dnf install VirtualBox
```

不过现在 VirtualBox 还没有完全准备好。它需要特殊的内核模块来运行，因为它必须具有对底层资源的访问决定权。Linux 内核里并不包含 VirtualBox 的内核驱动。在 Linux 源码树之外管理 Linux 内核驱动和使用默认安装内核驱动的方式相比始终是不那么方便。

VirtualBox 的内核驱动能够以编译源码模块的形式安装。这个过程可以用 `dkms` 命令自动化，当安装新的内核时，它会按照需求重新编译驱动。另一种简单而且不易出错的方式是根据你的操作系统发行版的使用已经编译好的内核模块。如果你的发行版已经提供了一个内核模块，它应该自动被加载。否则，你可以尝试 `modprobe vbxdrv` 命令。对于有些发行版，你可以像下面这样调用一个 `init.d` 的脚本来编译驱动：

```
sudo /etc/init.d/vboxdrv setup
```

既然 Vagrant 依赖都已经安装完成，我们可以启动一个 Vagrant 虚拟机。

下面的命令会根据模板生成一个 Vagrant 的配置文件。我们之后会修改这个文件。基础镜像是基于 Ubuntu 的 `hashicorp/precise32`。

```
vagrant init hashicorp/precise32
```

现在，我们可以启动机器：

```
vagrant up
```

如果一切顺利，我们的 `vagrant` 虚拟机应该已经开始运行了。它没有界面，所以我们什么都看不到。

Vagrant 和 Docker 有相似的地方。Docker 使用可以扩展的基础镜像。Vagrant 也是这样。在 Vagrant 的字典中，一个基础镜像被称为 **box**。

为了连接到先前启动的 `vagrant` 实例，我们可以用下面的命令：

```
vagrant ssh
```

现在我们有了 `ssh` 会话，可以通过它在虚拟机上工作了。为了实现这点，Vagrant 完成了一些任务，比如为我们设置 SSH 通信的密钥。

Vagrant 还提供了一个配置管理系统，这样可以完全根据源代码，通过 Vagrant 虚拟机描述符来重新创建一台虚拟机。

下面是一个处于早期阶段的 Vagrant 文件。为了简洁起见已经移除了注释。

```
Vagrant.configure(2) do |config|
  config.vm.box = "hashicorp/precise32"
end
```

在 Vagrant 文件中添加一行以调用我们提供的 bash 脚本：

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise32"
  config.vm.provision :shell, path: "bootstrap.sh"
end
```

脚本 `bootstrap.sh` 的内容如下：

```
#!/usr/bin/env bash
apt-get update
apt-get install -y apache2
```

这段脚本会在 Vagrant 管理的虚拟机上安装一个 Apache 服务器。

现在我们对 Vagrant 有了足够的了解，可以从 DevOps 的角度来评价它：

- Vagrant 是一种主要为基于 VirtualBox 的虚拟机方便管理配置的方式。从测试的角度来看用处很大。
- 配置的方法没有真的扩展到集群，而且这也不是期望的使用场景。
- 从另一个角度来说，几种配置管理系统如 Ansible 都支持 Vagrant，所以 Vagrant 在测试这些配置代码的时候比较有用。

用 Docker 做部署

关于部署的一个最新替代方案是 Docker，它有好几个非常有趣的特性。在本书中我们已经用过几次 Docker 了。

你可以利用 Docker 的特性来测试自动化脚本，即便你使用 Puppet 或者 Ansible 去部署产品。

Docker 可以用来创建可重用于开发机器、测试环境和产品环境的容器，这个模型非常

吸引人。

在撰写本书时，Docker 开始在大的企业中产生了影响力，但是类似 Puppet 这样的解决方案仍然占大多数。

大家都知道如何使用 Puppet 和 Ansible 去构建大的服务器集群，但是如何构建基于 Docker 的大型服务器集群并不那么众所周知。有几种新的解决方案，如：

- **Docker Swarm**：Docker Swarm 兼容 Docker Compose，这点很吸引人。Docker Swarm 由 Docker 社区维护。
- **Kubernetes**：Kubernetes 基于 Google 的 Borg 集群软件模型，其诱人之处在于所使用的模型经过了 Google 庞大的数据中心检验。Kubernetes 和 Borg 并不相同，这点要记住。目前尚不清楚的是 Kubernetes 是否提供了和 Borg 一样的伸缩能力。

对比表

大家都喜欢用新的名词去描述旧的概念。不同产品中的概念不尽相同，因此人们乐意去做出这样的配置系统的不同术语相互映射的字典。

以下是术语对照表：

System	Puppet	Ansible	Pallet	Salt
Client	Agent	Node	Node	Minion
Server	Master	Server	Server	Master
Configuration	Catalog	Playbook	Crate	Salt State

还有技术对照表：

System	Puppet	Ansible	Pallet	Chef	Salt
Agentless	No	Yes	Yes	Yes	Both
Client dependencies	Ruby	Python, sshd, bash	sshd, bash	Ruby, sshd, bash	Python
Language	Ruby	Python	Clojure	Ruby	Python

云计算解决方案

首先，我们得退回一步看看局势。我们既可以使用云计算提供商，比如 AWS 或者 Azure，

也可以使用自己内部的云计算解决方案，如 VMware 或者 OpenStack。根据企业的不同，可以使用外部或内部的云计算解决方案甚至同时使用。

有些类型的企业，比如政府部门，需要存储管理的群众的所有数据。这样的企业无法使用外部的云服务提供商和服务，必须建立相同功能的内部云服务。

小一点的私有企业可能会从使用外部的云计算提供商获益，但也许不能负担所有的资源都来自这样的提供商。它们可能会选择在负载较低时使用自己的服务器，负载达到峰值的时候再扩展到外部的云计算提供商。

我们这里介绍的很多配置管理系统都支持云平台节点和本地节点的管理。比如，PalletOps 支持 AWS，Puppet 支持 Azure。Ansible 支持不同云服务的主机。

AWS

Amazon Web Services 允许我们在 Amazon 的集群上部署虚拟机镜像。按照下面的步骤去设置 AWS：

1. 在 AWS 上注册一个账户。注册免费，但是需要一个信用卡号，即便你只使用免费资源。
2. 有一些身份验证的步骤，可以通过自动响应的电话来完成。
3. 当用户身份验证完成后，你就可以登录到 AWS，使用它的 web 控制台。

在我看来，AWS web 控制台的界面可用性一般，但是它能够完成任务。主机的选择有很多种，在我们的应用场景下更倾向于使用虚拟机和 Docker 容器。

4. 来到 **EC2 网络和安全**的部分。这里你可以创建后面所需要的管理的秘钥。

   作为第一个例子，让我们创建 AWS 提供的容器默认例子 console-sample-app-static。要登录到生成的服务器，首先你需要创建 SSH 秘钥对，并将公钥传给 AWS。一路单击下去你会得到一个小的范例集群。最后的资源创建步骤可能比较慢，所以到了去拿一杯咖啡的时候了。

5. 现在，我们可以查看集群的细节并且选择 web 服务器的容器。你能看到 IP 地址。尝试通过浏览器打开它。

既然我们有了可以工作的 AWS 账户，就可以用你选择的配置管理系统去管理它了。

Azure

Azure 是微软的云平台。它可以托管 Linux 和 Microsoft 虚拟机。虽然大家一般会选择 AWS 的服务，至少是在使用 Linux 的时候，尝试别的选项也无伤大雅。Azure 就是其中的一个选项，它目前正在慢慢地获得市场份额。

在 Azure 上创建虚拟机和在 AWS 创建虚拟机的体验类似。整个过程很顺畅。

总结

在本章中，我们探索了很多部署构建的代码的可用工具。选择有很多，这是有原因的。部署是一个很难的主题，你可能得花很多时间搞清楚哪个最适合你。

在下一章中，我们将会探索的主题是监控运行中的代码。

8

监控代码

在上一章中，我们探讨了部署代码的方式。

既然代码已经被你选择的部署解决方案安全地部署到了服务器上，你需要监控它以确保它正常运行。你可以花大量的时间准备在开发阶段设想的很多失败的模型。可是最后，你的软件会因为意料之外的其他原因失败。如果你的系统出问题，对企业来说是一件昂贵的事情，要么丢失利润，要么丢失信誉导致最终丢失利润。你需要在最快时间内知道哪些地方出现问题从而能够及时处理。

鉴于服务宕机的潜在负面影响，也有许多替代解决方案从不同的角度和观点处理监控已部署代码的问题域。

本章将会学习几种对我们来说可用的解决方案。

Nagios

在这个部分，我们探讨使用 Nagios 监控服务器整体运行状况的解决方案。

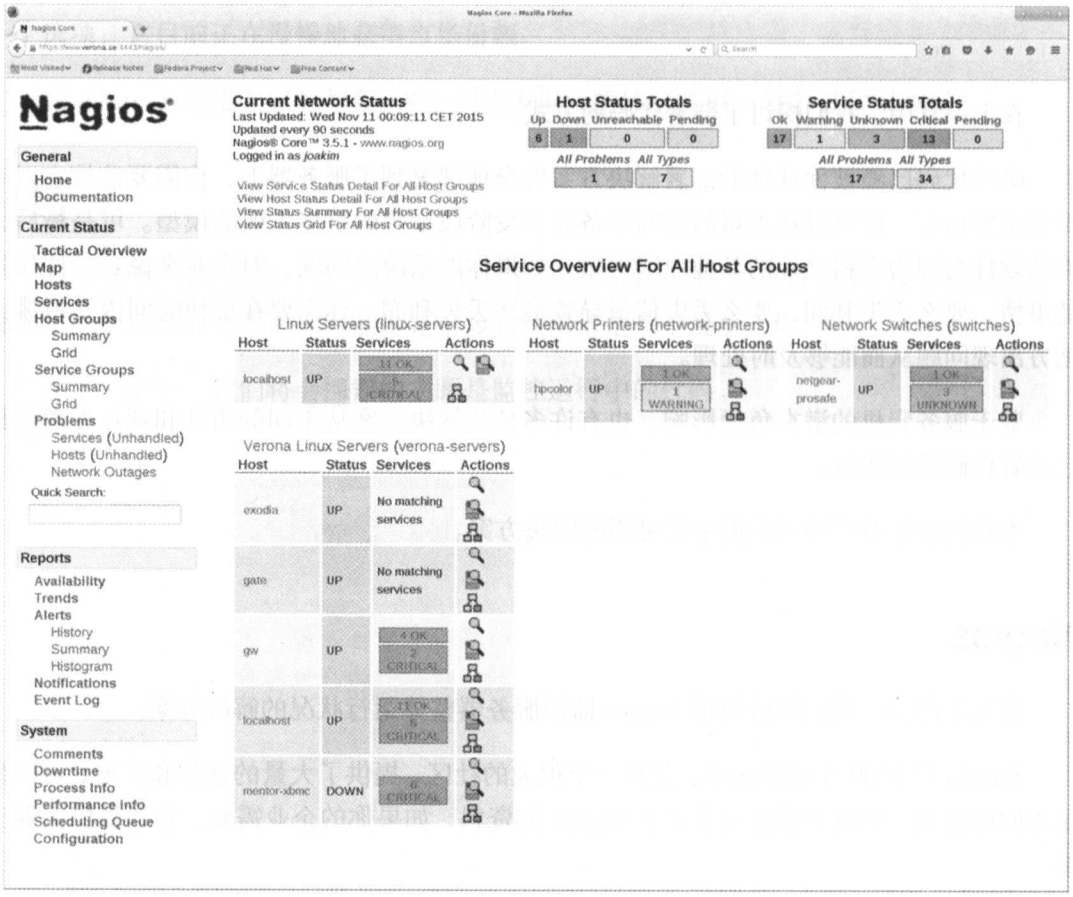
Nagios 自 1999 年就出现了，它有一个很大的社区，提供了大量的插件和扩展，可以满足不同的需求。网络上也有很多关于 Nagios 的资料，如果你的企业需要，它还提供商业支持。

因为存在了很长时间并且很多企业都在使用它，Nagios 已经成为了与其他解决方案对比时的网络监控的标准。因此，将它作为我们进入监控领域的旅程起点很正常。

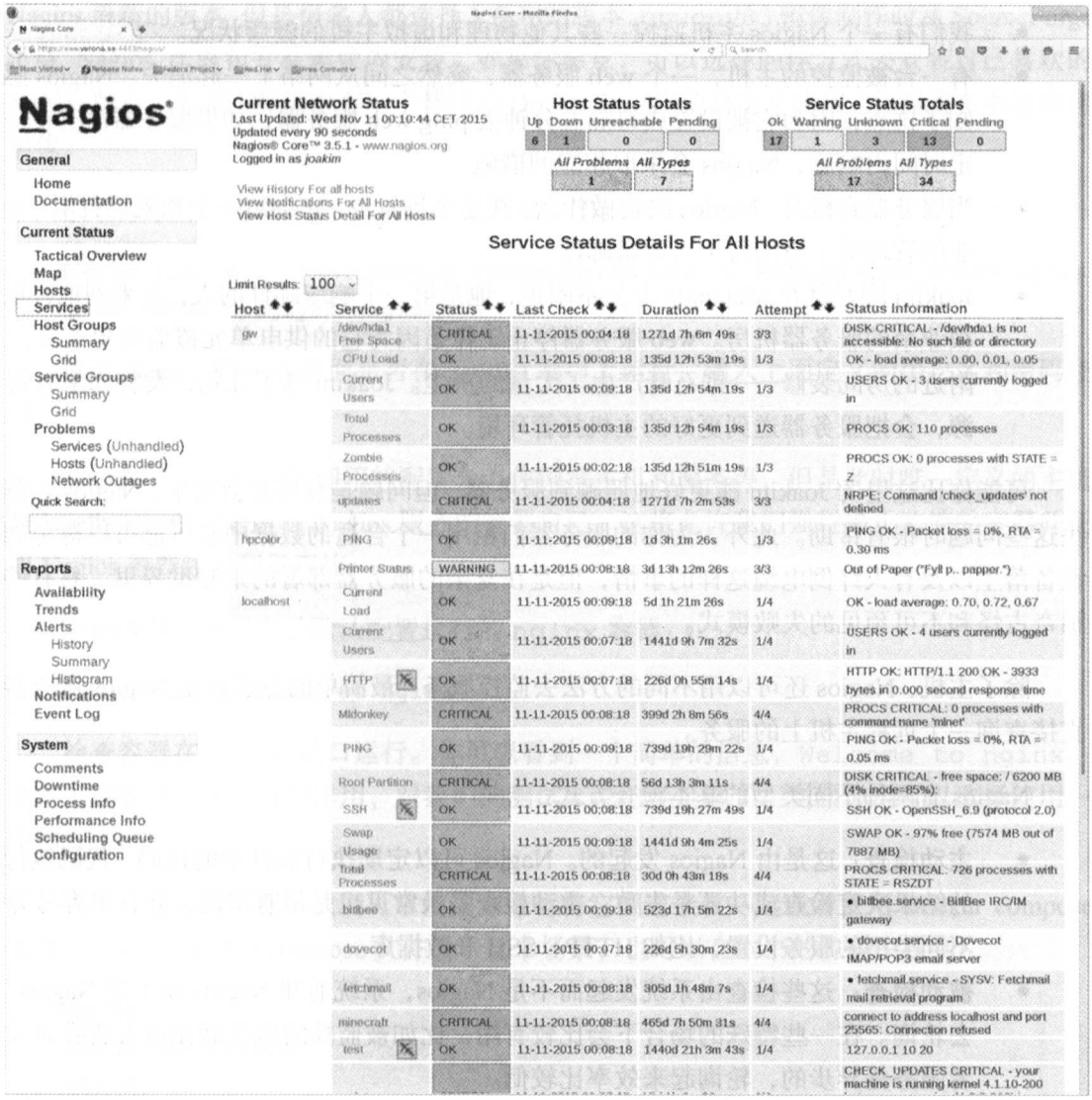
Nagios 这个名字是递归首字母缩写，在黑客圈里这样起名字是一个传统。它代表“Nagios Ain't Gonna Insist On Sainthood”。这个产品原本的名字叫作 NetSaint，但是由于商标的问题这个名字被拒绝了，因而得名 Nagios。agios 的希腊文意思是天使，所以整体来看这是一个很聪明的缩写。

下面是一些来自 Nagios 的小型演示的截图。Nagios 提供了很多视图，以下是其中的两个：

- 所有主机组的服务概览：



- 所有主机的服务状态细节：



Nagios

General
Home
Documentation

Current Status
Tactical Overview
Map
Hosts
Services
Host Groups
Summary
Grid
Service Groups
Summary
Grid
Problems
Services (Unhandled)
Hosts (Unhandled)
Network Outages
Quick Search:

Reports
Availability
Trends
Alerts
History
Summary
Histogram
Notifications
Event Log

System
Comments
Downtime
Process Info
Performance Info
Scheduling Queue
Configuration

Current Network Status
Last Updated: Wed Nov 11 00:10:44 CET 2015
Updated every 90 seconds
Nagios® Core™ 3.5.1 - www.nagios.org
Logged in as *joakim*

View History For all hosts
View Notifications For All Hosts
View Host Status Detail For All Hosts

Host Status Totals

Up	Down	Unreachable	Pending
6	1	0	0

All Problems	All Types
1	7

Service Status Totals

Ok	Warning	Unknown	Critical	Pending
17	1	3	13	0

All Problems	All Types
17	34

Service Status Details For All Hosts

Limit Results: 100

Host	Service	Status	Last Check	Duration	Attempt	Status Information
gw	/dev/hda1 Free Space	CRITICAL	11-11-2015 00:04:18	1271d 7h 6m 49s	3/3	DISK CRITICAL - /dev/hda1 is not accessible: No such file or directory
	CPU Load	OK	11-11-2015 00:08:18	135d 12h 53m 19s	1/3	OK - load average: 0.00, 0.01, 0.05
	Current Users	OK	11-11-2015 00:09:18	135d 12h 54m 19s	1/3	USERS OK - 3 users currently logged in
	Total Processes	OK	11-11-2015 00:03:18	135d 12h 54m 19s	1/3	PROCS OK: 110 processes
	Zombie Processes	OK	11-11-2015 00:02:18	135d 12h 51m 19s	1/3	PROCS OK: 0 processes with STATE = Z
	updates	CRITICAL	11-11-2015 00:04:18	1271d 7h 2m 53s	3/3	NRPE: Command 'check_updates' not defined
hpcolor	PING	OK	11-11-2015 00:09:18	1d 3h 1m 26s	1/3	PING OK - Packet loss = 0%, RTA = 0.30 ms
	Printer Status	WARNING	11-11-2015 00:08:18	3d 13h 12m 26s	3/3	Out of Paper ("Fyll p.. papper.")
localhost	Current Load	OK	11-11-2015 00:09:18	5d 1h 21m 26s	1/4	OK - load average: 0.70, 0.72, 0.67
	Current Users	OK	11-11-2015 00:07:18	1441d 9h 7m 32s	1/4	USERS OK - 4 users currently logged in
	HTTP	OK	11-11-2015 00:07:18	226d 0h 55m 14s	1/4	HTTP OK: HTTP/1.1 200 OK - 3933 bytes in 0.000 second response time
	Mldonkey	CRITICAL	11-11-2015 00:08:18	399d 2h 8m 56s	4/4	PROCS CRITICAL: 0 processes with command name 'mlnet'
	PING	OK	11-11-2015 00:09:18	739d 19h 29m 22s	1/4	PING OK - Packet loss = 0%, RTA = 0.05 ms
	Root Partition	CRITICAL	11-11-2015 00:08:18	56d 13h 3m 11s	4/4	DISK CRITICAL - free space: / 6200 MB (4% inode=85%):
	SSH	OK	11-11-2015 00:08:18	739d 19h 27m 49s	1/4	SSH OK - OpenSSH_6.9 (protocol 2.0)
	Swap Usage	OK	11-11-2015 00:09:18	1441d 9h 4m 25s	1/4	SWAP OK - 97% free (7574 MB out of 7887 MB)
	Total Processes	CRITICAL	11-11-2015 00:08:18	30d 0h 43m 18s	4/4	PROCS CRITICAL: 726 processes with STATE = RSZDT
	bitlbee	OK	11-11-2015 00:09:18	523d 17h 5m 22s	1/4	• bitlbee.service - BitlBee IRC/IM gateway
	dovecot	OK	11-11-2015 00:07:18	226d 5h 30m 23s	1/4	• dovecot.service - Dovecot IMAP/POP3 email server
	fetchmail	OK	11-11-2015 00:08:18	305d 1h 48m 7s	1/4	• fetchmail.service - SYSV: Fetchmail mail retrieval program
	minecraft	CRITICAL	11-11-2015 00:08:18	465d 7h 50m 31s	4/4	connect to address localhost and port 25565: Connection refused
	test	OK	11-11-2015 00:08:18	1440d 21h 3m 43s	1/4	127.0.0.1 10 20
						CHECK_UPDATES CRITICAL - your machine is running kernel 4.1.10-200

这个样例视图揭示了如下一些错误：

- 一台主机访问不可达。
- 某台主机的一块硬盘快满了。
- 一台主机挂起了可用更新。

现在让我们用一个简单的场景来查看 Nagios 的基础以及通用监控：

- 我们有一个 Nagios 主机监控一些其他物理和虚拟主机的健康状况。
- 有一台被监控的主机，一个 web 服务器，突然之间从网络上一起消失。Nagios 感知了这点，因为它被设置成每隔五分钟去 ping web 服务器，如果服务器没有在一定时间内响应，Nagios 会采取相应的措施。
- 当服务器宕机时，Nagios 决定做什么，在这个场景下，它会给一个预先设置组（企业的管理员）里的每个人发送邮件。
- Joakim 刚好就在 Matangle 办公室附近，他是第一个响应邮件的人，并来到地窖里很业余的服务器机房。web 服务器停止工作是因为它的供电单元覆盖灰尘太多，附近的房间装修——锯石膏产生了大量的灰尘。Joakim 骂了几句，发誓如果有资源，会把服务器送到更好的主机托管环境。

可能你比这里的 Joakim 能更好地照顾到服务器，但问题是意外总会发生，Nagios 在解决这些问题时很有帮助。此外，为你的服务器们租用一个合适的数据中心可能可以避免如设备落尘以及有人绊倒电缆这样的事情，但是在复杂的服务器部署的未知世界里，总有些稀奇古怪和不可预见的失败模式。

除了主机，Nagios 还可以用不同的方法去监控服务。最简单的方法就是 Nagios 服务器直接查询一个远程主机上的服务。

Nagios 有两种不同类型的基本检查方式：

- **主动检查**：这是由 Nagios 发起的。Nagios 可以定期执行插件中的代码。代码执行的结果决定检查成功还是失败。主动检查是最常见也是最有用的，并且很容易为不同类型的服务设置，比如 HTTP、SSH 和数据库。
- **被动检查**：这些检查由系统发起而不是 Nagios，系统通知 Nagios 而不是 Nagios 去轮询。在一些特殊的场合下会比较有用，比如被监控的服务器在防火墙后面或者服务是异步的，轮询起来效率比较低。

如果你已经习惯了基于文件的配置，那么配置 Nagios 是比较直接的，但即使为了获得基本的功能还是需要很多的配置工作。

我们将使用 Docker Hub 的镜像 `cpuguy83/nagios` 来尝试 Nagios（译者注：这里的镜像已经不可用，可以替换为这个镜像 `jasonrivers/nagios`）。

镜像使用的 Nagios 版本是 3.5，在撰写本书时，和 Fedora 仓库中的版本一致。这不是 Nagios 最新的版本，但是很多人都选择 3 系列的版本。cpuguy83 镜像制作时从 SourceForge 下载 Nagios 压缩包并在镜像内安装。如果你愿意，可以选择同样方式去安装自己喜欢的 Nagios 版本。本书的源码中有一个替代的 Dockerfile，以便自定义镜像，你需要在本地构建镜像。

以下命令可以启动一个 Nagios 服务器的容器：

```
docker run -e    NAGIOSADMIN_USER=nagiosadmin -e NAGIOSAMDIN_PASS=nagios
-p 80:30000 cpuguy83/nagios
```

验证 Nagios 的 web 界面是否在 30000 端口上运行，输入上面定义的用户名和密码登录。

Nagios 用户界面开箱即用的配置允许你浏览主机和服务等，但是此时唯一定义的主机是 localhost，也就是运行 Nagios 服务的容器自己。这将允许我们强行停止监控的容器并证明 Nagios 容器注意到了事故。

下面的命令可以启动默认配置运行的 nginx 容器：

```
docker run -p 30001:80 nginx
```

检查容器在 30001 端口运行。你可以看到一个简单的信息，Welcome to nginx。基本上这就是这个容器的作用，对我们的测试来说已经足够了。当然你也可以选择使用一个物理主机。

为了一起运行容器，我们可以使用命令行将它们连接在一起或者使用 Docker compose 文件。下面的 Docker compose 文件适于这个场景，也可以在本书的源码中找到：

```
nagios:
 image: mt-nagios
 build:
- mt-nagios
ports:
  - 80:30000
 environment:
   - NAGIOSADMIN_USER=nagiosadmin
   - NAGIOSAMDIN_PASS=nagios
```

```
  volumes:
   ./nagios:/etc/nagios
 nginx:
  image: nginx
```

Nagios 配置以 Docker 卷的方式挂载在./nagios 目录下。Nagios 需要监控的配置也在源码中，具体内容如下：

```
define host {
    name regular-host
    use  linux-server
    register  0
    max_check_attempts   5
}

define host{
    use   regular-host
    host_name   client1
    address    192.168.200.15
    contact_groups  admins
    notes          test client1
}
hostgroups.cfg
define hostgroup {
    hostgroup_name  test-group
    alias          Test Servers
    members         client1
}

services.cfg
#+BEGIN_SRC sh
define service {
  use           generic-service
  hostgroup_name  test-group
  service_description PING
  check_command   check_ping!200.0,20%!600.0,60%
}
```

让我们来试试如果 nginx 容器宕机会发生什么事情。

用 docker ps 命令找到 nginx 容器的哈希码。然后用 docker kill 命令干掉它。再次用 docker ps 命令验证这个容器已经消失。

现在，等待一会后，刷新 Nagios web 用户界面。你应该可以看到 Nagios 已经对这个情况报警了。

这和前面显示错误服务的截图很相似。

现在，当 NGINX 宕机时你应该能收到一封邮件。然而，确保这个例子万无一失地工作并没有这么简单，因为垃圾邮件的原因，邮件要更加复杂些。你需要知道你的邮件服务器的细节，比如 SMTP 服务器细节。以下是你需要填写的具体细节的模板：

你可以创建一个名为 contacts.cfg 的文件去处理邮件，文件内容如下：

```
define contact{
    contact_name    matangle-admin
    use             generic-contact
    alias           Nagios Admin
    email           pd-admin@matangle.com
}

define contactgroup{
    contactgroup_name   admins
    alias               Nagios Administrators
    members             matange-admin
}
```

如果你不修改这个配置，邮件会被发送到 pd-admin@matangle.com，并被当成垃圾邮件忽略掉。

Munin

Munin 用来描绘服务器统计数据如内存使用，对于理解服务器整体的健康很有帮助。Munin 按照时间去描绘服务器统计数据，所以你可以看到资源分配的趋势，在情况变得糟糕前帮助你找到问题。还有几个类似 Munin 的绘图工具，但是和 Nagios 一样，Munin 也是一个很好的学习起点。

它的设计易于使用和配置。开箱即用的体验让你可以用很小的代价绘制很多图像。

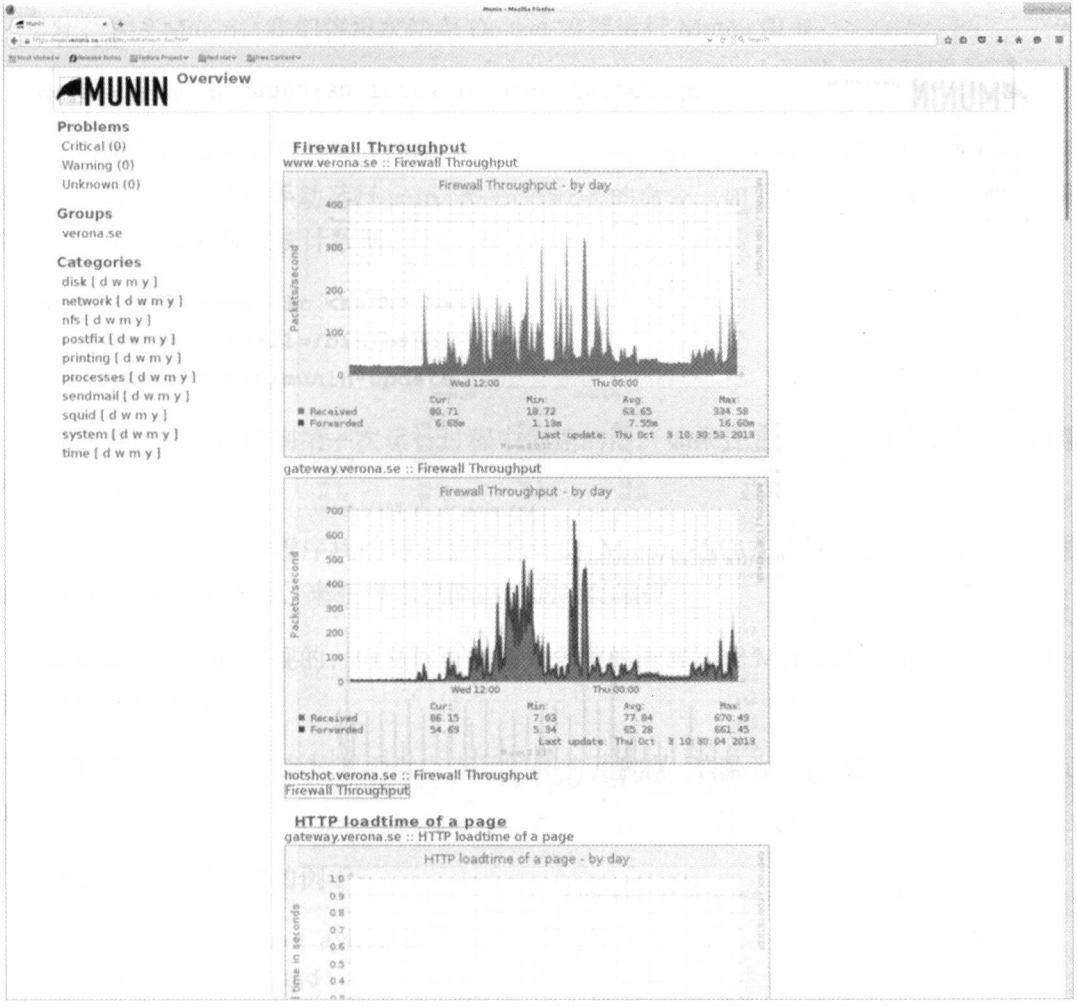
在北欧神话传说中，Hugin 和 Munin 是两只漆黑的乌鸦。它们不知疲倦地飞到尘世中收集信息。长途旅行结束后，它们回到神王 Odin 的肩上，告诉它所有收集到的信息。Hugin 这个词源于思考，Munin 源于记忆。

Nagios 着重于服务健康的高级别特性（不论服务或者主机在线或者没有运行），Munin 周期性地抽样，持续跟踪获得的统计数据并绘图。

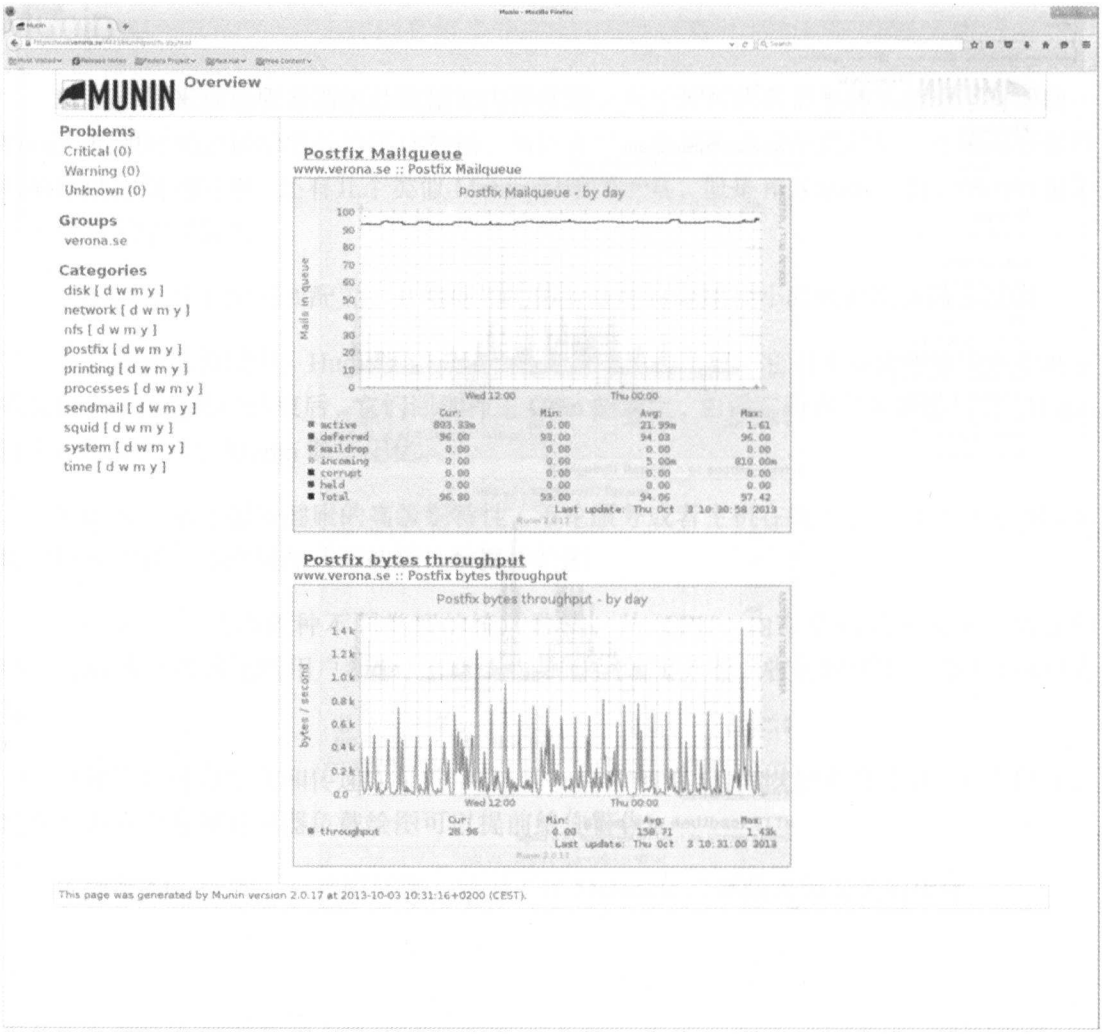
Munin 可以处理各种不同类型的统计数据，从 CPU、内存负载到你家孩子所在的 Minecraft 服务器的活跃用户数量。想要定制从自己的服务器上收集数据统计的插件也很容易。

一张图片可以包含和传递大量的信息，看一眼就知道了，所以如果将要出现什么问题，对服务器的内存和处理器负载绘图可以提前给你警告。

下面是 Munin 的一个截图样例，展示了在 Matangle 总部防火墙安装的指标：



这里是企业的 smtpd 一些指标，Postfix 的安装：



和我们在本章前面探讨过的任何监控系统类似，Munin 也有一个面向网络的架构。在设计方面和 Nagios 相似。Munin 的主要组件如下：

- Munin master 是一个中心化服务器，负责从 Munin 的节点收集数据。Munin 的数据保存在叫作 **RRD** 的数据库系统中，RRD 是**轮询数据库（Round-robin Database）**的缩写。RRD 同时也做收集数据的绘图。
- Munin 节点是被监控的服务器上的一个组件。Munin master 连接所有的 Munin 节点，运行插件返回数据给 master。

为了尝试 Munin，我们将再次使用 Docker 容器运行 Munin 服务：

```
docker run -p 30005:80 lrivallain/munin:latest
```

第一次运行 Munin 需要一点时间，所以，在浏览 web 界面前先稍等一下。如果你不喜欢等待，可以在容器中手动运行 `munin-update` 的命令，如下所示。它会显式地轮询所有的 Munin 节点来获取统计数据。

```
docker run exec -it <hash> bash
su - munin --shell=/bin/bash
/usr/share/munin/munin-update
```

现在你应该可以看到第一次运行过程中创建的图。如果让它运行一段时间，你可以看到图是如何发展的。

实现一个监控应用程序栈的特定数据统计的 Munin 插件并不困难。你可以编写一段 shell 脚本让 Munin 调用，来获得你想要追踪的统计数据。

Munin 是用 Perl 实现的，但是你可以用大多数的语言实现 Munin 的插件，只要遵从一个简单的接口即可。

程序使用 `config` 参数调用时应当返回一些元数据。这是为了让 Munin 在图上标出正确的标签。

下面是一个图配置的例子：

```
graph_title Load average
graph_vlabel load
load.label load
```

发送数据只要打印到 `stdout` 即可。

```
printf "load.value "
cut -d' ' -f2  /proc/loadavg
```

下面这段脚本可以绘制出机器的平均负载：

```
#!/bin/sh

case $1 in
```

```
    config)
        cat <<'EOM'
graph_title Load average
graph_vlabel load
load.label load
EOM
    exit 0;;
esac

printf "load.value "
cut -d' ' -f2  /proc/loadavg
```

这个系统很简单也很可靠，你也能很容易为自己的应用实现它。需要做的只是将你的数据统计打印到 stdout 即可。

Ganglia

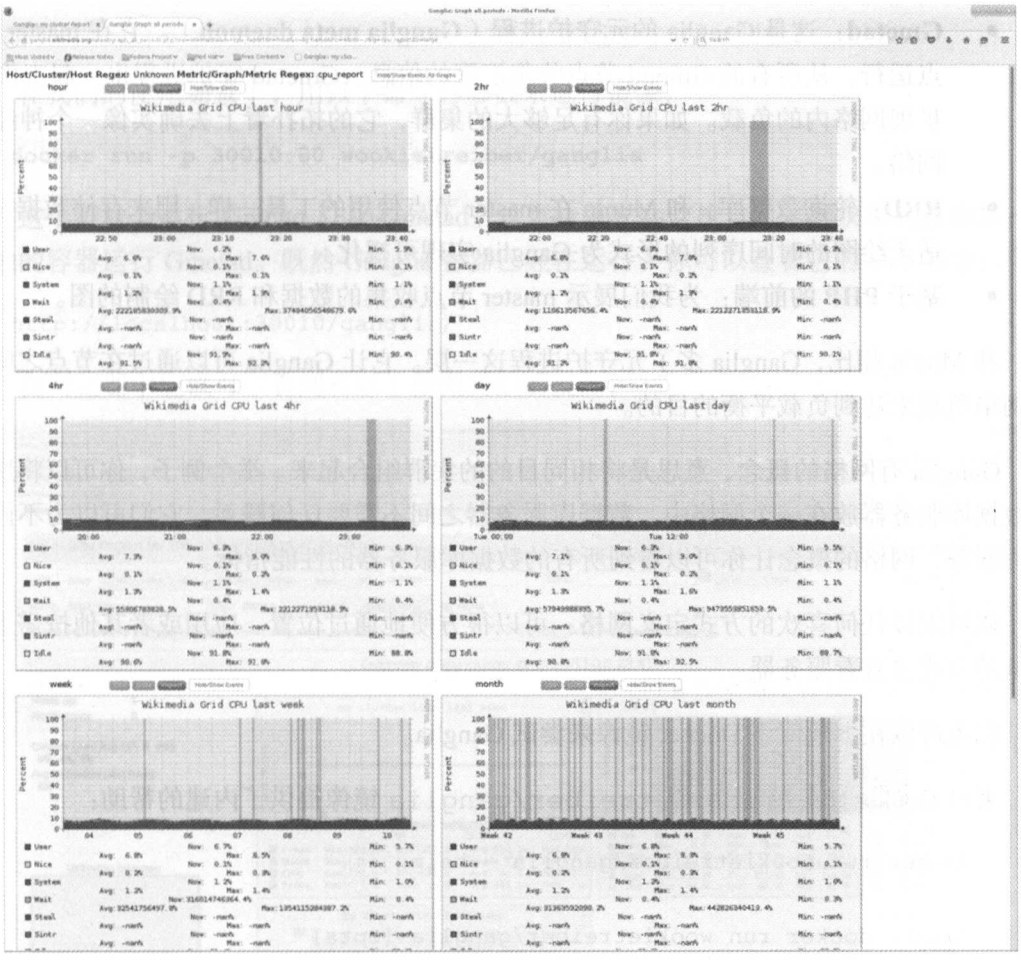
Ganglia 是大型集群的绘图和监控解决方案。它可以汇总信息并以简单的形式展示出来。

“ganglia”是 ganglion 的复数形式，神经学中神经细胞群的意思。这意味着 Ganglia 可以成为你集群的感觉神经元网络。

和 Munin 一样，Ganglia 也使用 RRD 作为数据库存储和绘图，所以图会和前面的 Munin 图类似。代码复用真是一件好事！

在 http://ganglia.wikimedia.org/latest/Ganglia 上有个很有趣的在线演示。

Wikimedia 为 Wikipedia 提供媒体内容服务，非常繁忙。演示很好地让我们看到了 Ganglia 功能概览，这样的功能你很难在自己的网络中轻松获得。



第一页展示了图格式中可用数据的概览。你可以在图表里向下展开以获取其他视图。

举个例子，如果你根据 CPU 负载展开应用集群的图，可以获得集群中每个独立服务器的详细视图。可以进一步单击服务器节点来获得更多信息。

如果你也有 Wikimedia 那种规模的集群，你可以很清楚地看到 Ganglia 概览和展开视图的吸引力。兼具全局视图和易于访问的细节等特点。

Ganglia 由下面的组件构成：

- **Gmond**：这是 **Ganglia 监控守护进程(Ganglia monitoring daemon)**的缩写。Gmond 是收集节点信息的一个服务。你需要在每个想让 Ganglia 监控的服务器上安装它。

- **Gmetad**：这是 Ganglia 的**元守护进程（Ganglia meta daemon）**。它在 master 节点运行，从所有的 Gmond 节点收集汇聚的信息。Gmetad 守护进程也一起工作来扩展网络内的负载。如果你有足够大的集群，它的拓扑看上去确实像一个神经元网络。
- **RRD**：轮询数据库，和 Munin 在 master 节点使用的工具一样，用来存储数据并以适于绘图的时间序列的形式为 Ganglia 实现可视化。
- **基于 PHP 的前端**：为我们展示 master 节点收集的数据和 RRD 绘制的图。

和 Munin 相比，Ganglia 多了元守护进程这一层。它让 Ganglia 可以通过在节点之间分散网络负载来达到负载平衡的目的。

Ganglia 有网格的概念，意思是将相同目的的集群组合起来。举个例子，你可以将所有的数据库服务器放在一个网格中。数据库服务器之间不需要任何联系，它们可以为不同的应用服务。网格的概念让你可以看到所有的数据库服务器的性能指标。

你可以以任何喜欢的方式定义网格：可以很方便地通过位置、应用或者其他按需逻辑分组的方式去查看服务器。

你也可以在本地使用 Docker 镜像来尝试 Ganglia。

来自 Docker Hub 的 `wookietreiber/ganglia` 镜像提供了内建的帮助：

```
docker run wookietreiber/ganglia --help

Usage: docker run wookietreiber/ganglia [opts]

Run a ganglia-web container.

   -? | -h | -help | --help      print this help
   --with-gmond                   also run gmond inside the
container
   --without-gmond                do not run gmond inside the
container
   --timezone arg                 set timezone within the
container,
                                  must be path below /usr/share/
zoneinfo,
```

```
e.g. Europe/Berlin
```

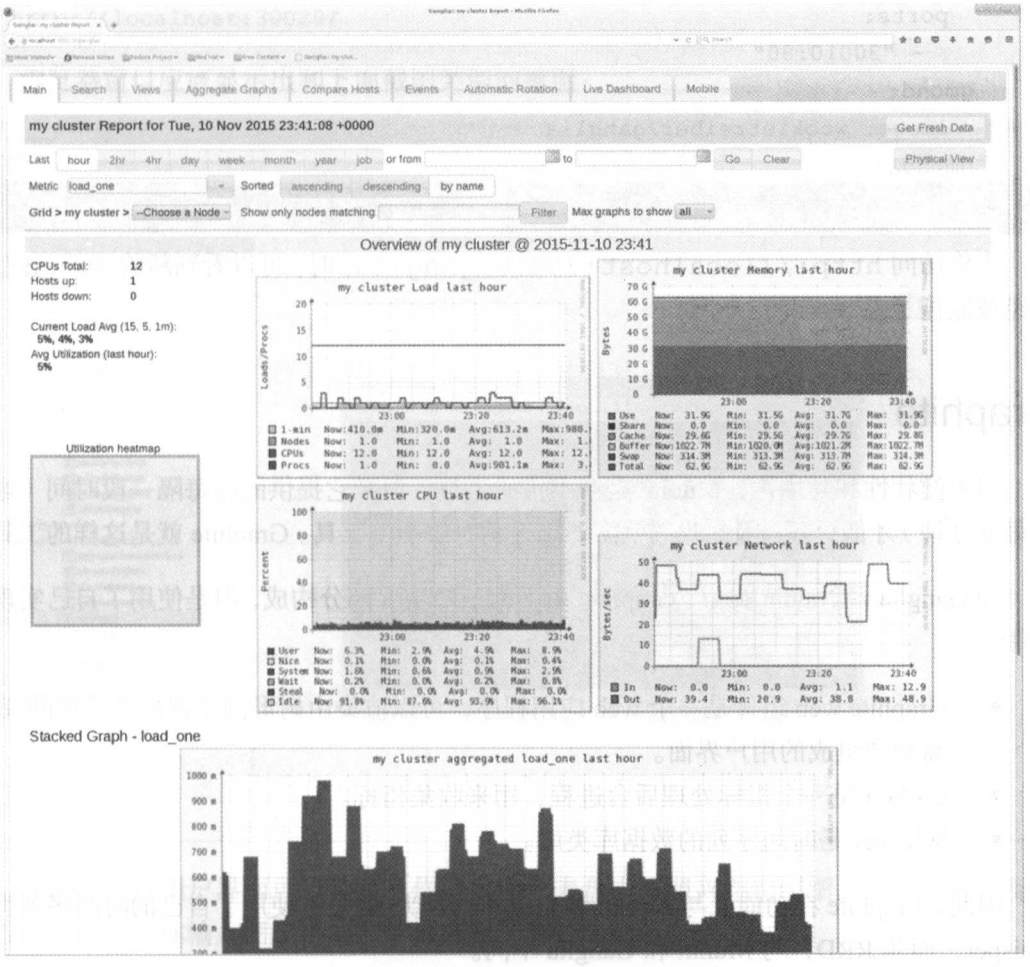
在我们的使用场景下，用以下命令行参数来运行镜像：

```
docker run -p 30010:80 wookietreiber/ganglia
```

这个镜像会运行 Gmond 以及 Gmetad。这意味着它会监控自己。很快我们会添加一个单独的容器运行 Gmetad。既然 Ganglia 容器已经在运行，你可以查看它的 web 界面：

```
http://localhost:30010/ganglia/
```

在浏览器中应该可以看到和以下截图类似的视图：



Ganglia Gmond 节点之间用一个多播的 IP 通道进行通信。这提供了在大型集群配置的冗余性和易用性，多播的配置是默认的。你也可以将 Ganglia 配置为单播模式。

因为我们只运行两个通信的容器，在这种情况下我们实际上不会从多播的配置中收益。

还有，我们会使用 Docker Compose 来启动所有容器，这样分开的 Gmond 和 Gmetad 实例可以相互通信。我们可以启动三个运行 gmond 的容器。Gmetad 守护进程在默认配置下会发现它们并且在 My Cluster 下面列出：

```
  gmetad:
    image: wookietreiber/ganglia
    ports:
      - "30010:80"
  gmond:
    image: wookietreiber/ganglia
  gmond2:
image: wookietreiber/ganglia
```

当你访问 http://localhost:30010/ganglia/时，可以看到新的 gmond 节点已经被监控了。

Graphite

因为健壮性和易用性，Munin 是个不错的工具。但是它提供的图每隔一段时间（通常是每五分钟）才能更新。因此我们就需要接近实时绘图的工具。Graphite 就是这样的工具。

和 Ganglia 及 Munin 类似，Graphite 由下面三个主要部分构成，但是使用了自己实现的组件。

- Graphite web 组件是一个 web 应用程序，可以渲染由树形浏览器控件管理的仪表盘和图组成的用户界面。
- Carbon 是一个指标处理后台进程，用来收集指标。
- Whisper 是时间序列的数据库类库。

因此，Graphite 在功能上与 Munin 和 Ganglia 类似。但是它使用了自己的时间序列类库 Whisper，而非 RRD，与 Munin 和 Ganglia 不同。

可以通过几个预先打好包的 Docker 镜像来尝试 Graphite。我们可以使用 Docker Hub 的 `sitespeedio/graphite` 镜像，命令如下：

```
docker run -it \ -p 30020:80 \ -p 2003:2003 \ sitespeedio/graphite
```

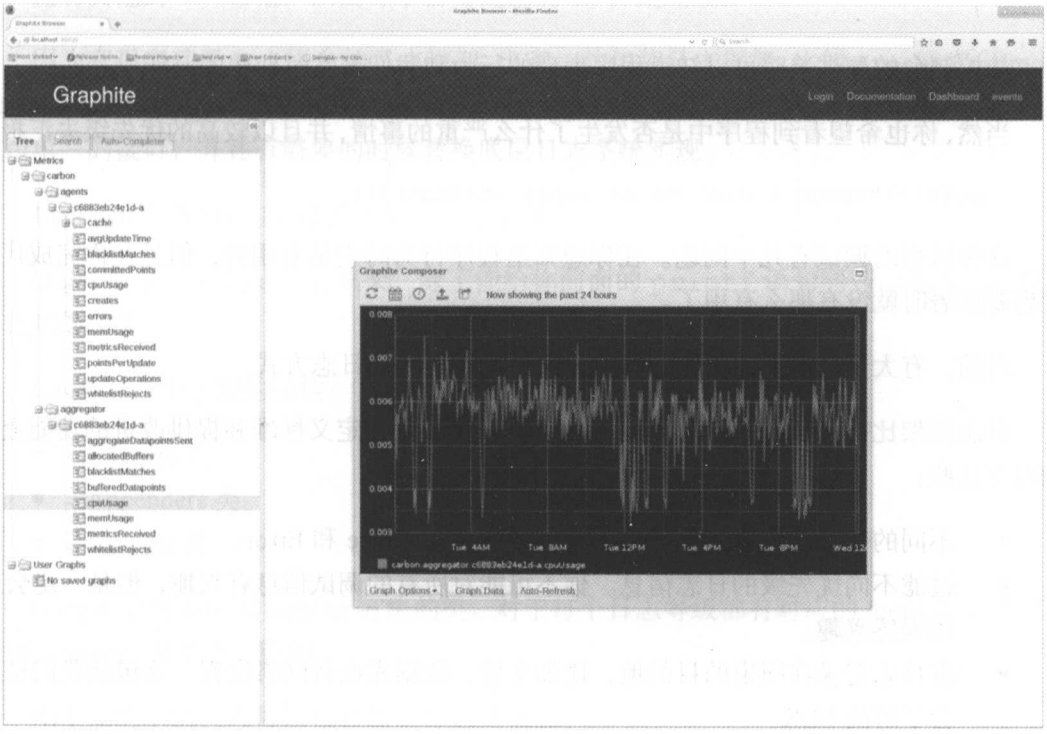
这会启动一个 Docker 容器，运行使用 HTTP 基本验证方式的 Graphite。

你现在可以查看 Graphite 的用户界面。用户名和密码都是“guest”。通过给 image 提供一个 `.httpasswd` 文件可以修改用户名和密码。

如果你没有修改过前面命令中的端口号，那么 URL 应该是：

```
http://localhost:30020/
```

浏览器窗口应该显示出和下面截图类似的视图：



Graphite 的用户界面可以定制，你可以在仪表盘中按照兴趣组织图。有一个开发完成的控件，可以让你输入图名字的首字母来找到需要的图。

日志处理

日志处理是一个很重要的概念，我们将尝试其中一些工具，比如 ELK（Elasticsearch、Logstash 和 Kibana）。

传统意义上的日志只是在代码中使用简单的打印语句来追踪代码中的事件。有时候这被叫作**打印式调试（printf-style debugging）**，因为跟踪代码行为而不是使用常规的调试器。

下面是一个 C 语言风格的简单例子。目的是为了知道当我们输入函数 *fn(x)*时，传入的参数 *x* 值是什么：

```
void fn(char *x){
  printf("DEBUG entering fn, x is %s\n", x);
  ...
}
```

从控制台的调试追踪可以知道程序表现的行为和我们开发时期望的一样。

当然，你也希望看到程序中是否发生了什么严重的事情，并且以较高的优先级去汇报：

```
printf("ERROR x cant be an empty string\n");
```

这种风格的调试有几个问题。在你想知道程序行为时它是有用的，但是在你完成代码后想要部署时就没有那么有用了。

目前，有大量基于上述实践考验的框架，支持不同的日志方式。

日志框架比打印风格的日志有更多的价值，尤其是在定义标准和提供改进的功能上，如以下这些：

- 不同的日志优先级，比如 Debug、Warning、Trace 和 Error。
- 过滤不同优先级的日志信息。你不可能对所有的调试信息有兴趣，但是一定会对错误感兴趣。
- 将日志记录在固定的目的地，比如文件、数据库或者网络进程。这包括我们稍后会了解的 ELK。
- 日志文件的轮替和归档。老的日志文件可以被归档。

时不时地就会有新的日志框架出现，所以即使在如今，日志问题领域看上去也还远远

没有到令人满意的地步。这种趋势是可以理解的，因为处理得当的日志可以帮助你准确判断一个不再运行的网络服务的失败原因，或者你不经常管理的复杂服务。日志也很难做好，因为过度的日志会降低服务的性能，日志太少又不能帮你诊断失败的原因。因此，日志系统尽力去在日志的不同特性间达到一个平衡。

客户端日志类库

Log4j 是一个流行的 Java 日志框架。有好几种语言都移植了这个框架，比如：

- 针对 C 语言的 **Log4c**。
- 针对 JavaScript 语言的 **Log4js**。
- 针对微软.NET 框架的 **Apache log4net**。

还有其他的一些移植，以及分享自身概念的不同日志框架。

因为有很多基于 Java 平台的日志框架，也有一些包装日志框架，比如 **Apache Commons Logging** 或者 **Simple Logging Facade for Java**（**SLF4J**）。这些框架的目的是为了让你可以使用单一的接口，并在有需要的时候替换底层日志系统实现。

Logback 为 log4j 的继任者而生，兼容 ELK。

Log4j 是第一个 Java 日志框架，除了能给你前面的 `printf` 语句同等的输出，还有更多花哨的功能。

Log4j 有三个主要的结构：

- Loggers 类。
- Appenders 类。
- Layouts 类。

Logger 是用来访问 logging 方法的类。对于每个日志等级都有对应的 looging 方法可以调用。Logger 也是分等级的。

用下面的示例代码可以很容易解释这些概念：

```
Logger  logger = Logger.getLogger("se.matangle");
```

这给了我们企业一个单独的 logger。这样我们就可以在由于使用其他企业的代码而生

成的一堆日志里，生成自己的日志信息。对于 Java 企业环境来说这点很有用，你可能有很多个类库都使用 log4j 记录它们的日志，并且你想要为不同的类库配置不同的日志等级。

为了更细粒度的日志我们可以同时使用几种 logger。不同的软件组件可以使用不同的 logger。

下面的例子来自一个视频录制应用：

```
Logger  videologger = Logger.getLogger("se.matangle.video");
logger.warn("disk is dangerously low")
```

我们可以使用几种不同的日志等级，为我们的日志增加更大的明确性：

```
videologger.error("video encoding source running out prematurely")
```

用 log4j 的术语来说，日志消息最后结尾的地方被称为 **appender**。有很多可用的 appender，比如控制台、文件、网络目标地址和 logger 后台进程。

布局控制我们的日志信息格式。这允许我们使用类似 printf 的转义序列格式。

比如，`PatternLayout` 配置使用的转换模式为`%r [%t] %-5p %c - %m%n`，会产生下面的输出：

```
176 [main] INFO  se.matangle - User found
```

下面是模式中每个字段代表的意思：

- 第一个字段是从程序开始运行后经过的时间，以微秒计算。
- 第二个字段是发起日志请求的线程。
- 第三个字段是日志语句的等级。
- 第四个字段是和日志请求关联的 logger 名字。
- 在“-”之后的文本是消息语句。

Log4j 努力让日志的配置暴露给外部的应用。这样开发人员就可以通过在文件或者控制台配置让日志在本地工作。随后，当应用被部署在产品服务器时，管理员可以将日志的 appender 配置为其他东西，比如我们后面要讨论的 ELK。这种方式不需要修改代码，而且我们在部署的时候可以修改日志的行为和目的地。

像 WildFly 这样的应用服务器通过实现 log4j 系统插件为自己提供配置系统。

如果你不使用应用服务器也没关系，log4j 的新版本支持很多不同的配置格式。下面是一个 XML 格式的文件，它会在 classpath 上寻找 `log4j2-test.xml` 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

ELK

ELK 由下面的组件构成：

- **Logstash**：这是一个和 log4j 类似的开源日志框架。Logstash 的服务器组件处理到来的日志。
- **Elasticsearch**：它保存所有的日志并且对其索引，顾名思义，是为了搜索。
- **Kibana**：是日志搜索和可视化的 web 界面。

为了知道它如何工作，我们会跟随一个日志信息，从它的代码出处，经过中间的网络层，一路到最终的网络操作者的屏幕上：

1. 在代码深处，发生了一个 Java 异常。应用意外的不能将从导入的文件中出现的用户 ID 映射到数据库中的用户 ID 中去。log4j 记录了下面的错误：

```
logger.error("cant find imported user id in database")
```

2. 2log4j 系统配置将错误日志记录到 Logstash 后端，如下所示：

```
log4j.appender.applog=org.apache.log4j.net.SocketAppender
```

```
log4j.appender.applog.port=5000
log4j.appender.applog.remoteHost=master
log4j.appender.applog.DatePattern='.'yyyy-MM-dd
log4j.appender.applog.layout=org.apache.log4j.PatternLayout
log4j.appender.applog.layout.ConversionPattern=%d %-5p [%t] %c %M
- %m%n
log4j.rootLogger=warn, applog
log4j.logger.nl=debug
```

3. Logstash 的系统配置如下，监听 log4j 描述的端口。Logstash 后台进程必须和你的应用分开启动：

```
input {
    log4j {
     mode => server
     host => "0.0.0.0"
     port => 5000
     type => "log4j"
  }
}
output {
stdout { codec => rubydebug }
stdout { }
  elasticsearch {
    cluster => "elasticsearch"
  }
}
```

4. Elasticsearch 引擎接受日志输出并且使其可以被搜索。

5. 管理员可以打开 Kibana GUI 应用实时看到日志的显示或是搜索历史数据。

你可能会认为这很复杂，确实是这样。但有趣的是，可以配置 log4j 来支持这个复杂的场景以及更加简单的场景，所以使用 log4j 或者它的兼容的竞争对手并不会真的出错。

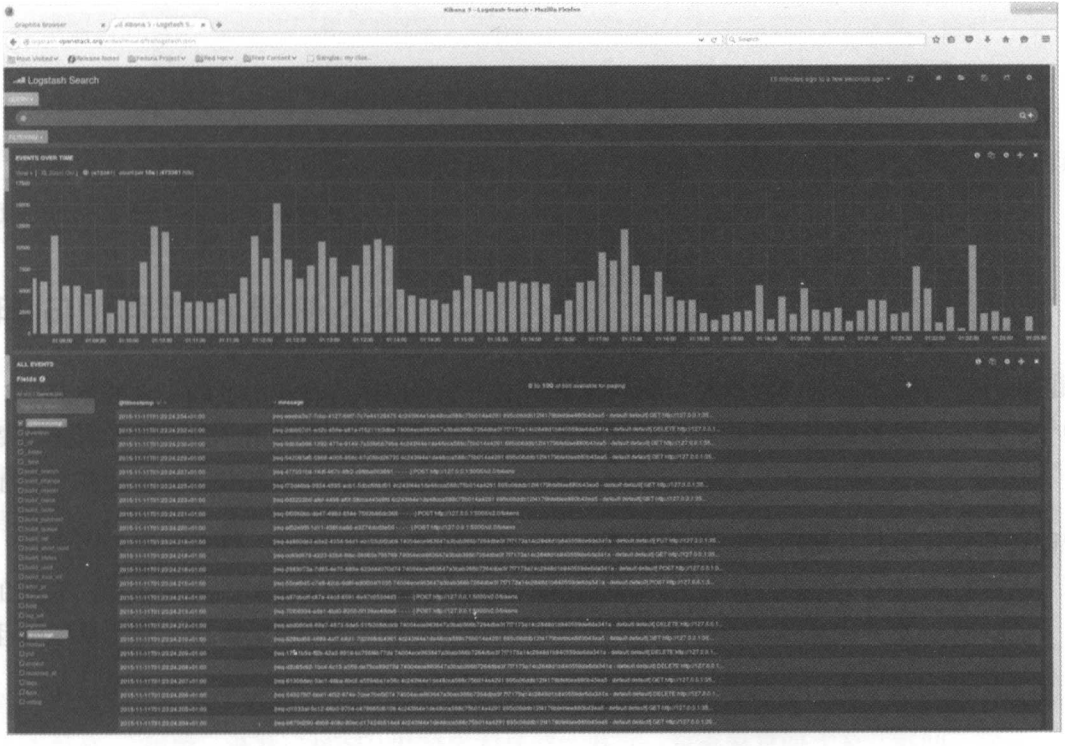
你可以一开始只用纯日志文件而不使用任何网络日志。这在很多情况下已经够用了。在你需要 ELK 额外的功能时，再去添加即可。保留日志文件也很有用。如果高级日志工具因为某些原因失败，大不了求助于 Unix 上传统的 `grep` 工具在日志文件里搜索。

Kibana 有很多可以帮助你分析日志数据的功能。你可以将数据绘制成图，为你寻找的特定模式过滤日志数据。

为了在实践中尝试这些，我们可以使用来自 Docker Hub 的 Kibana 和 Elasticsearch 的官方镜像：

```
docker run -d elasticsearch &&
docker run --link some-elasticsearch:elasticsearch -d kibana
```

如果一切顺利，我们应该可以访问 Kibana 用户界面，看起来像下面这样：



总结

本章我们学习了监控已部署代码的一些工具：监控主机和服务状态的 Nagios，从我们的主机绘制统计数据的 Munin、Ganglia 和 Graphite，以及跟踪日志信息的 log4j 和 ELK。

在下一章中，我们会了解帮助企业进行开发的工作流工具，比如问题跟踪器。

9

问题跟踪

在上一章，我们看到了如何通过监控和日志处理方案来监控已部署的代码。

在本章，我们将会看到企业内部处理开发流程的系统，例如问题跟踪软件。这样的系统是敏捷流程中很重要的帮手。

用问题跟踪器做什么？

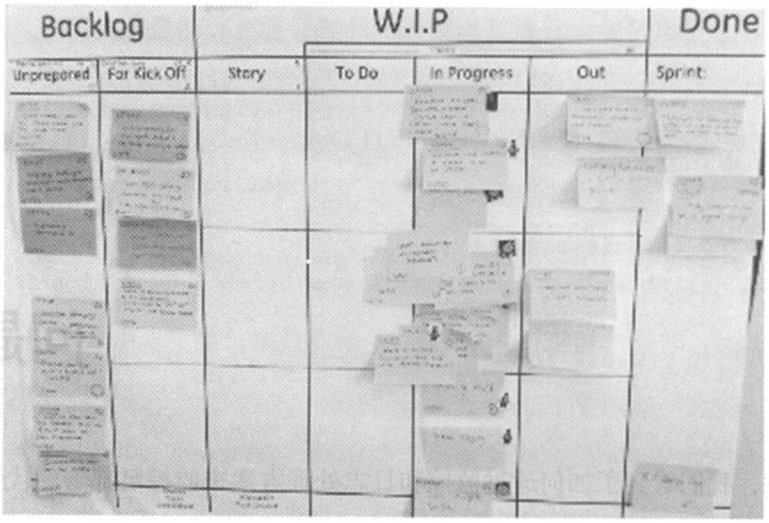
从敏捷流程的立场上看，问题跟踪器用来帮助处理敏捷流程的各种细枝末节。问题跟踪器处理的可以是工作细目、缺陷和问题。大多数敏捷流程都包含着在物理板上或电子板通过便利贴管理任务的概念。

当用敏捷方法时，一般会有一块问题板，上面贴着手写的便利贴。这是看板方法的核心观念，因为看板在日语里表示招牌（signboard）。这块板直观地展示了工作流程概览，并且很容易管理，因为你只要移动便利贴就可以表示当前流程的状态发生变更。

改变看板也非常容易，只要用记号笔重画就可以了。例如你可以画上如下图的道道。

物理板在 Scrum 团队中也很流行。

另一方面，大多数基于 web 的数字化问题跟踪器，能为远程办公提供更为详尽的信息。这些问题跟踪器还能帮你记住流程的步骤。



理想情况下，人们会喜欢物理板和问题跟踪器，但是保持它们同步需要大量的工作。没有简单的办法能够解决这个问题。有些人用投影仪播放问题跟踪器使其看起来就像一块物理板那样，但是它并没有相同的触感，并且在投影图像或显示器移动任务与在物理板上并不一样。物理板还有一个优势是它总是健壮不宕机的，并且当团队成员想要讨论任务时参考起来很方便。

技术上，问题跟踪器还是比较复杂的，通常都是以数据库对象代表状态机的方式实现的。通过 web 界面，用诸如电子邮件或自动化的方式创建一个问题。接下来，人们与之互动使其达到各种状态。到最后，问题被归档为关闭状态，以便未来参考。有时，流程的推移基于系统的交互。举个简单的例子：任务由于过期而自动关闭，所以就失效了。

工作流和问题的一些例子

尽管“问题”这个词贯穿本章，它在一些系统里称作“凭证”或是“缺陷”。在技术上它们是同一样东西。一个问题可能是一个待办事项、功能需求或是其他类型。一项功能改进和一个缺陷在技术上根本是同一样东西，可能有些反直觉；不过如果你视功能改进为功能缺失，那就开始能感觉到那么点意思了。

一个问题有许多的元数据，取决于它表示的意思和问题跟踪器的支持。最基本的问题类型很简单，也非常有用。它有如下的这些基本属性：

- **描述**：这是关于问题的文本描述，形式自由。
- **报告者**：这表明了谁创建了这个问题。
- **指派**：这是应该工作在这个项目上的人。

另外，它有两种状态：打开和关闭。

一般来说，这是问题跟踪器所提供的最基本功能。如果你把这个同便利贴相比，唯一多出的元数据是报告者，即使删了它跟踪器很可能仍然可用。在便利贴方式上，通过把便利贴粘贴到板上或者取下来就能轻松处理打开和关闭的状态。

当项目经理们开始使用电子问题跟踪器时，通常会热衷于往状态机和跟踪器的存储里增加复杂性。也就是说，很明显有许多信息可以很自然地被添加到问题跟踪器里。

除了前面提到的基本信息以外，这些额外的属性一般是很有用的：

- **到期日**：预期问题被解决的日期。
- **里程碑**：里程碑是一种将数个问题归并为一个较大工作包的方法。例如，一个里程碑可以表示成一个 Scrum 的 sprint。里程碑通常也有一个到期日，如果你使用里程碑功能，一般就不再使用各个单独问题的到期日了。
- **附件**：能够给问题添附截图和文档是很方便的，当开发者开始工作或是测试者验收时都可能会很有帮助。
- **工作量估计**：对解决问题的耗时有个估算会很有用。这可以用来做计划。同样，问题的真实耗时也会对各种计算有帮助。另一方面，做估算经常很棘手，最终的成本可能比估算本身能带来的价值还高。许多团队在他们的问题跟踪器里不用这类信息，也能工作得很好。

下面还有一些比打开和关闭更有用的状态，可以用来建模敏捷工作流。为了更加明确，我们再次把打开和关闭状态列在这里：

- **打开**：问题被报告，暂时还未指派人在上面工作。
- **进行中**：有人被指派到这个问题上，正在尝试解决它。
- **可以测试**：问题修复，现在需要验证。又是未指派。
- **测试中**：有人被指派到这个问题上，正在测试。
- **完成**：任务标记为已完成，再次未指派。完成状态被用来标记问题以持续跟踪它们，例如对 Scrum 方法而言，就意味着 sprint 结束。

- **关闭**：这个问题不再被关注，但是仍然留有记录以备日后参考。

在最佳的场景里，问题有序地从一个状态切换到下一个状态。在更加现实的场景里，很可能由于测试可能发现了问题并没有真正被解决，导致了一个任务从测试中回到了打开状态，而并非是完成。

我们需要从问题跟踪器里得到什么？

除了支持先前描述的基本工作流程外，我们还需要从问题跟踪器里得到什么？有不少令人关注的点，其中的一些并不那么明显。下面列出一些：

- 需要处理的数量？

  许多工具在 20 个用户以内的时候表现良好，一旦超过，就不得不考虑性能和许可证的需求。我们需要能跟踪多少数量的问题？有多少用户需要访问问题跟踪器？这些是可能会遇上的问题。

- 需要多少许可证？

  对于这个问题来说，免费软件完胜，因为收费软件的价格可能会吓人一跳。免费软件可以不用付费，而在需要支持时收费。

  除了 Jira 以外，本章提到的大部分问题跟踪器都是免费软件。

- 有性能限制吗？

  性能通常不是一个限制因素，因为大部分的跟踪器都使用适合于生产环境的数据库，诸如用 PostgreSQL 或是 MariaDB 作为后端数据库。大多数本章提到的跟踪器在企业内部使用的数量级上表现良好。Bugzilla 已经被证明可以处理大量的问题并且面向公共网络。

  然而，估算你想要部署系统的性能是一个好的实践。你想要的某些特性可能会导致一些性能问题。例如，大多数问题跟踪器在后台使用一个关系型数据库，而在你需要递归查询树形结构时，关系型数据库并不是最佳选择。在正常的用例里并不算是个问题，但是如果你想要在大数量级上使用深度嵌套树，问题可能就会浮出水面。

- 有什么可选的支持，质量如何？

  在真正使用之前确定支持的质量很难，但是有一些思路可以帮助评估：

  - 对于开源项目来说，支持取决于用户社区的大小。另外，一般都有可用的商业支持提供者。
  - 商业问题跟踪器可以既有付费支持，又有社区支持。

- 我们能不能使用外部的问题跟踪器？

  有许多公司提供问题跟踪器。包括 Atlassian 的 Jira，它可以由客户自行安装或者由 Atlassian 托管。还有一个托管问题跟踪器的例子是来自 Trello 公司的 Trello。

  部署问题跟踪器到你的内网基本不会太复杂。在安装复杂度上，本书涉及到的系统普遍都比较简单。 通常你需要一个数据库后台和 web 应用程序后台层。一般来说，最难的部分是与认证服务器集成，还有邮件服务器。然而，尽管安装自己的问题跟踪器绝非难事，使用托管的跟踪器更是轻而易举。

  有些企业由于法律或其他原因无法将工作数据置于外网。对这些企业来说，托管的问题跟踪器并不是可选项。他们只能在自己内网部署问题跟踪器。

- 问题流程系统满足我们的需求吗？

  一些系统，比如 Jira，允许高扩展性状态机来定义流程和集成编辑器。其他的更多是最简流程。你的需求决定了正确的方案。有些人从一个非常复杂的流程着手，最终发现其实只需要打开和关闭的状态。其他人认识到他们需要复杂的工作流来支持复杂的流程。

  一个可配置的流程对于大型企业是很有帮助的，因为它能将不那么明显的流程封装为知识。例如，修复一个缺陷时，质量保证部门的 Mike 需要验证。在一个小企业里没那么有用，因为每个人都知道要做些什么。然而，大型企业截然不同。能帮忙决定谁将处理接下来的任务并做些什么是很有用的。

- 问题跟踪器支持我们选择的敏捷方法吗？

  本章提到的大多数问题跟踪器的基本数据模型都足够灵活，可以代表任何类型的敏捷流程。从某种意义上说，你真正的需求是一个可配置的状态机。当然，问题

跟踪器可能对特定的敏捷方法和流程提供额外的功能以更好地支持。

这些支持敏捷方法的额外功能一般都源于两种类型：可视化和报表。虽然这些都属锦上添花的功能，但我觉得大家还是对它们过分关注了，尤其是对经验不足的组长来说。可视化和报表当然能提供价值，但是并没有像初见时认为的那样多。

在选择自己的问题跟踪器时，记住这些。确认你是否真的想要用那些令人印象深刻的报表。

- 这个系统与企业的其他系统集成是否容易？

  问题跟踪器可能要集成的系统包括代码库、单点登录、电子邮件系统等。例如，一个失败的构建可能会使构建服务器在问题跟踪器上自动生成一个凭证（ticket），并将其分配给破坏构建的开发者。

  因为这里主要关注开发流程，我们最需要的集成是代码库系统。这里讨论的大多数问题跟踪器都提供某些方式可以集成代码库系统。

- 跟踪器是否可扩展？

  如果问题跟踪器有作为扩展点的 API，那通常是很有用的。API 可以用来集成到其他系统，也可以定制跟踪器使其符合企业的流程。

  也许你需要把现在处于打开状态的问题基于 HTML 展示在公用显示器上。这可以通过一个好的 API 来实现。也许你想要问题跟踪器来触发部署系统的部署。这也可以通过一个好的问题跟踪器 API 来实现。

  一个好的 API 能够提供许多可能性。

- 跟踪器是否提供多项目支持？

  如果你有许多团队和项目，为每一个项目提供单独的问题跟踪器可能会很有用。因此，问题跟踪器的一项有用功能是能够分开数个相互隔离的子项目。

  同样，需要一些权衡。如果你有许多隔离的问题跟踪器，如何把一个问题从其中一个工具移到另一个？当你有几个团队工作在不同的任务集上时会需要这样的功

能。例如，开发团队和质量保证团队。从 DevOps 的视角来看，工具使团队相互远离而非靠近并非好事。

所以，虽然每个团队在主系统里拥有自己的问题跟踪器是一件好事，从整体上看能够支持处理不同团队间的任务也是必要的。

- 问题跟踪器是否支持多客户端?

  虽然这项功能并不是选择问题跟踪器的重要需求，但是如果能够从各种客户端和界面访问问题跟踪器还是很方便的。例如，开发者能够在 IDE 里直接访问问题跟踪器是非常方便的。

  一些工作在免费软件上的企业喜欢一个完整的基于电子邮件的流程，例如 Debian 的 debugs。尽管这样的需求在企业中比较稀有。

问题跟踪器激增所带来的问题

由于现在配置一个问题跟踪器在技术上没什么难度，团队经常会自行配置问题跟踪器来处理自己的问题和任务。这也在许多其他类型的工具上发生，如编辑器，但是编辑器只是开发者的个人工具，并且它们的主要作用并不是分享或与其他人协作。因此问题跟踪器的激增是一个问题，而编辑器的激增并不是什么问题。

引起问题跟踪器激增的一个原因是：大型企业可能会标准化某种类型的跟踪器，而没有几个人喜欢用它。一个典型的原因是，当决定问题跟踪器的时候，只考虑到了特定团队的需求，例如质量保证团队或者运维团队。另一个原因是只购买了有限的问题跟踪器许可证，而团队的剩余成员就只能自己凑合着用了。

还有一种常见的场景是开发者用一种跟踪器，质量保证团队用另一种，而运维团队用了第三种不能兼容的系统。

所以，虽然企业作为一个整体用许多不同的问题跟踪器来处理相同的事情是一个次优解，但是这也可以当作一个团队在这个领域开始选择适合自己工具的一个胜利。

再说，DevOps 的核心思想是将不同的团队和角色紧密连接起来。互相不兼容的协作工具的激增并不是件好事。在大型企业里经常会发生这样的事，而对于这个问题并没有简单

的解决方案。

使用一个免费的跟踪器软件，至少可以摆脱有限许可证问题的困扰。找到一个能在各方面都取悦每个人的系统当然绝非易事。在限制选择范围上，让真正天天与之打交道者满意的问题跟踪器会更有帮助，而非在管理上有帮助的那些。通常这意味着减少对漂亮的燃尽图的关注，而更多关注在真正地做完事情上。

所有的跟踪器

接下来，我们将会探索一组不同的问题跟踪器。它们都非常容易试用，方便在你做决定前参考。大多数都是免费的，但是也会涉及到一些收费的替代方案。

这里提到的所有的跟踪器都可以在维基百科的比较页面上找到：`https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems`。

鉴于只能探索有限的问题跟踪器，我们选择了基于不同设计而不同的问题跟踪器。Bugzilla 是被设计来支持面向公众的大型跟踪器。Redmine 是一个全功能项目管理工具的例子。GitLab 由于简单并集成了 Git 而被选中，还有因为易用性而选择的 Jira。

我们从 Bugzilla 开始问题跟踪器的探索之旅，因为它是早期还很流行的问题跟踪器之一。这意味着它的那些流程都非常成熟，因为它们在足够长的时间内已经证明了自己。

Bugzilla

Bugzilla 可以说是本章所有的问题跟踪器的教父。Bugzilla 自 1998 年创建以来，一直被许多高姿态的企业使用，例如 Red Hat、Linux 内核和 Mozilla。如果曾经报告过这些项目的缺陷，你很可能已经接触到 Bugzilla 了。

Bugzilla 是由 Mozilla 维护的免费软件，Mozilla 也出品了著名的 Firefox 浏览器。Bugzilla 是用 Perl 语言编写的。

顾名思义，Bugzilla 关注的是缺陷而非其他的问题类型。如果需要其他类型的任务，例如提供一个 wiki，需要单独来处理。

许多企业把 Bugzilla 作为一个面向公网的问题报告工具。千锤百炼之下，Bugzilla 拥有很不错的安全性。

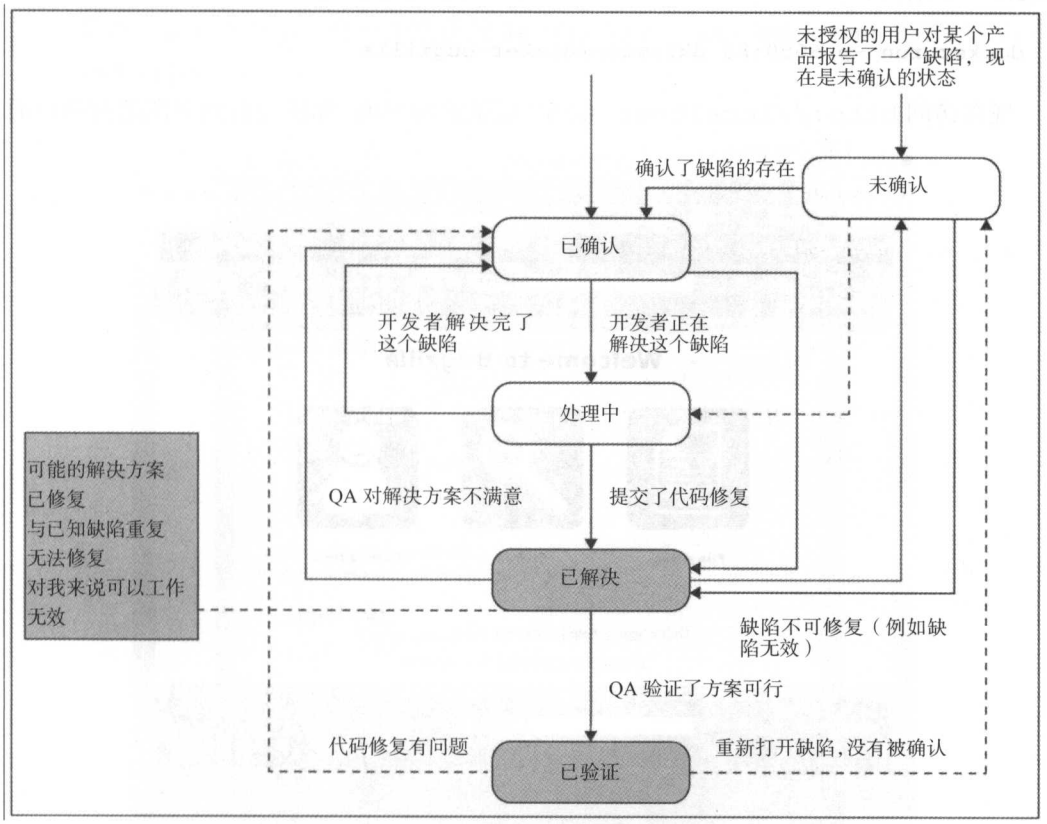
在 https://landfill.bugzilla.org/bugzilla-tip/有一个已安装的演示版，你可以测试最新的 Bugzilla 开发版的功能。

Bugzilla 支持自定义。可以给问题增加自定义字段并自定义工作流程。

Bugzilla 提供了 XML-RPC 和 JSON REST API。

由于存在了很长时间，Bugzilla 有许多的扩展程序和插件，包括可选的客户端。

下图展示了 Bugzilla 的缺陷状态机，或者说是工作流程：



这个流程是可配置的，它的默认状态如下：

- **未确认**：来自未授权用户的缺陷始于未确认状态。
- **已确认**：如果认为缺陷值得调查，它就被置于已确认状态。
- **处理中**：当一个开发者开始解决缺陷，就使用这个状态。

- **已解决**：当开发者认为已经完成了缺陷修复，他们把缺陷置于已解决状态。
- **已验证**：当质量保证团队认可缺陷已被修复，它就被置于已验证状态。

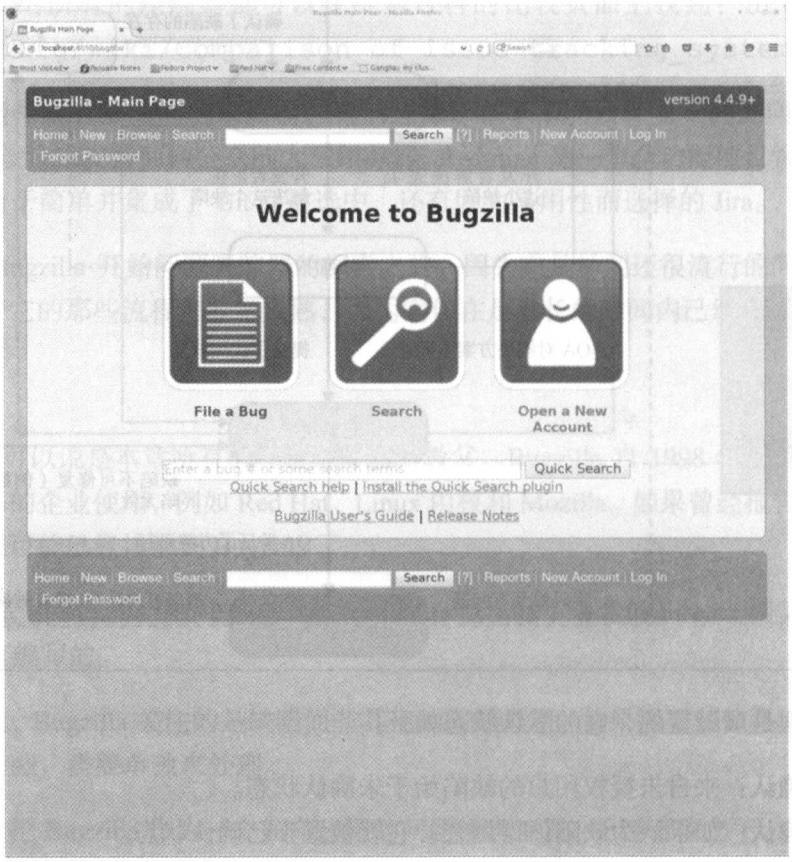
现在让我们开始试用 Bugzilla。

在 Docker hub 上有一些 Bugzilla 的镜像可用，在你的 Linux 发行版的库里很可能也会有可用的 Bugzilla。

以下命令可以在单独的容器里启动 Bugzilla，以及一个完整的配置适当的 MySQL。随便选择一个端口：

```
docker run -p 6050:80 dklawren/docker-bugzilla
```

现在访问 `http://localhost:6050` 试试它的界面。你将会看到下图的欢迎页面：

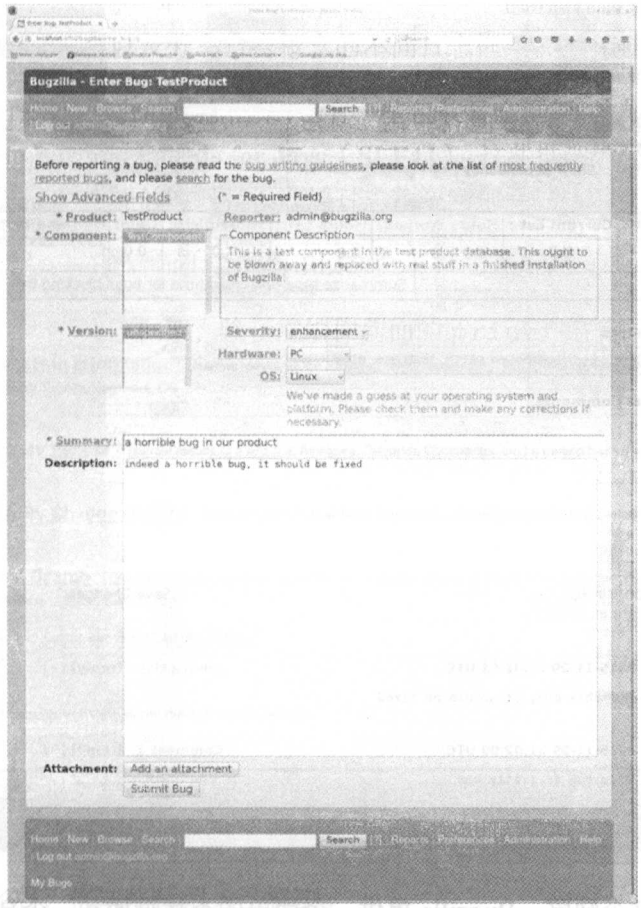


Bugzilla 要求你有一个用于提交缺陷的账号。可以使用管理员账号或者创建一个新账号。

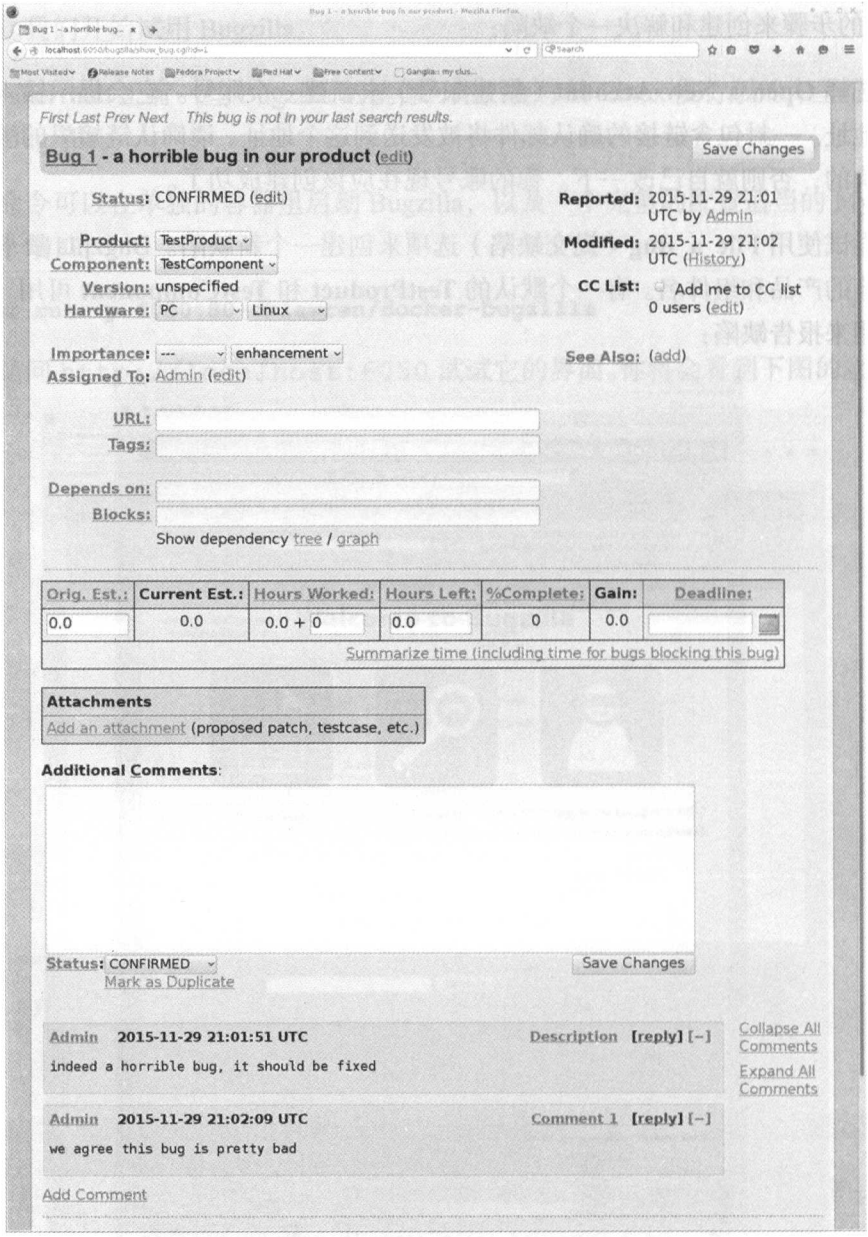
可以用管理员用户名 `admin@bugzilla.org` 来登录，默认密码是 `password`。现在按下面的步骤来创建和解决一个缺陷：

1. 选择 **Open a New Account（新建账号）**来新建一个账号。它会提示输入电子邮件地址。一封包含链接的确认邮件将被发送到这个地址。请确认链接里的端口号是正确的，否则就自己改一下。你的账号现在应该创建成功了。

2. 尝试使用 **File a Bug（提交缺陷）**选项来创建一个新缺陷。Bugzilla 要求你提供缺陷的产品和组件名。有一个默认的 **TestProduct** 和 **TestComponent** 可用，我们可以用来报告缺陷：

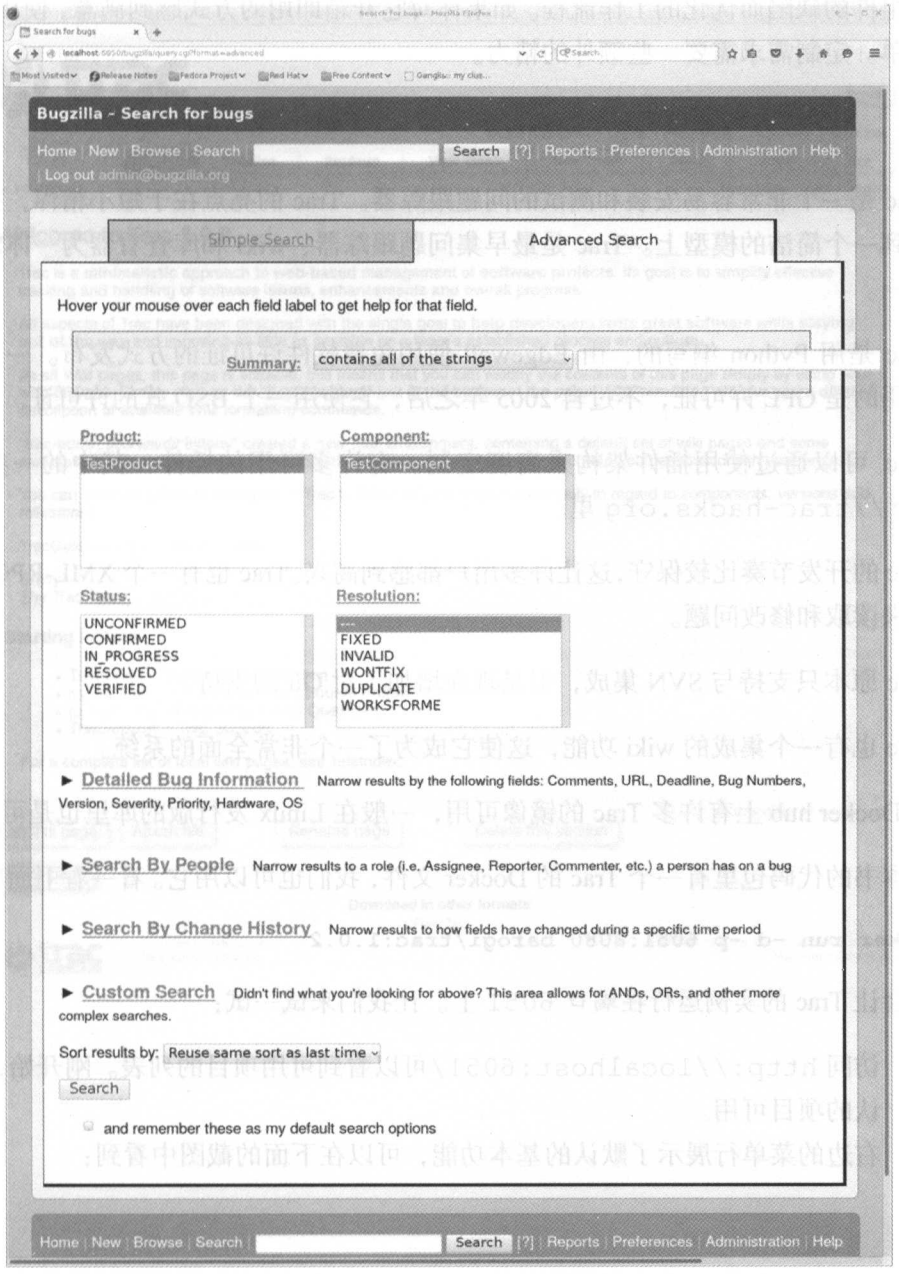


3. 现在缺陷处于打开状态，我们可以加点备注。视操作权限级别而定，当在缺陷里写备注时，我们也能把它移动到一个新状态上。尝试像下图那样添加一些备注：



4. 尝试查找这个缺陷。Bugzilla 提供一个简单的查找页面和一个高级的查找页面。高

级的页面如下图所示，它允许你使用大部分的字段组合：



5. 最终完成之后，把缺陷标记为已解决。

如你所见，Bugzilla 拥有我们希望从问题跟踪器里获得的所有功能。它更适用于为大量产品和组件的缺陷而优化的工作流程。如果你对它开箱即用的方式感到满意，使用 Bugzilla 非常简单。定制需求需要一些额外的精力。

Trac

Trac 是一个非常容易安装和测试的问题跟踪器。Trac 的亮点在于短小精悍，把一些系统集成到一个简洁的模型上。Trac 是最早集问题跟踪器、wiki 和库查看器为一体的问题跟踪器之一。

Trac 是用 Python 编写的，由 Edgewall 使用免费软件许可证的方式发布。一开始的时候它采用的是 GPL 许可证，不过自 2005 年之后，它使用一个 BSD 式的许可证。

Trac 可以通过使用插件架构来高度定制。有许多可用的插件，其中的一些列在了 `https://trac-hacks.org` 里。

Trac 的开发节奏比较保守，这让许多用户都感到高兴。Trac 也有一个 XML-RPC 的 API，可以用来读取和修改问题。

Trac 原本只支持与 SVN 集成，但是现在增加了对 Git 的支持。

Trac 也有一个集成的 wiki 功能，这使它成为了一个非常全面的系统。

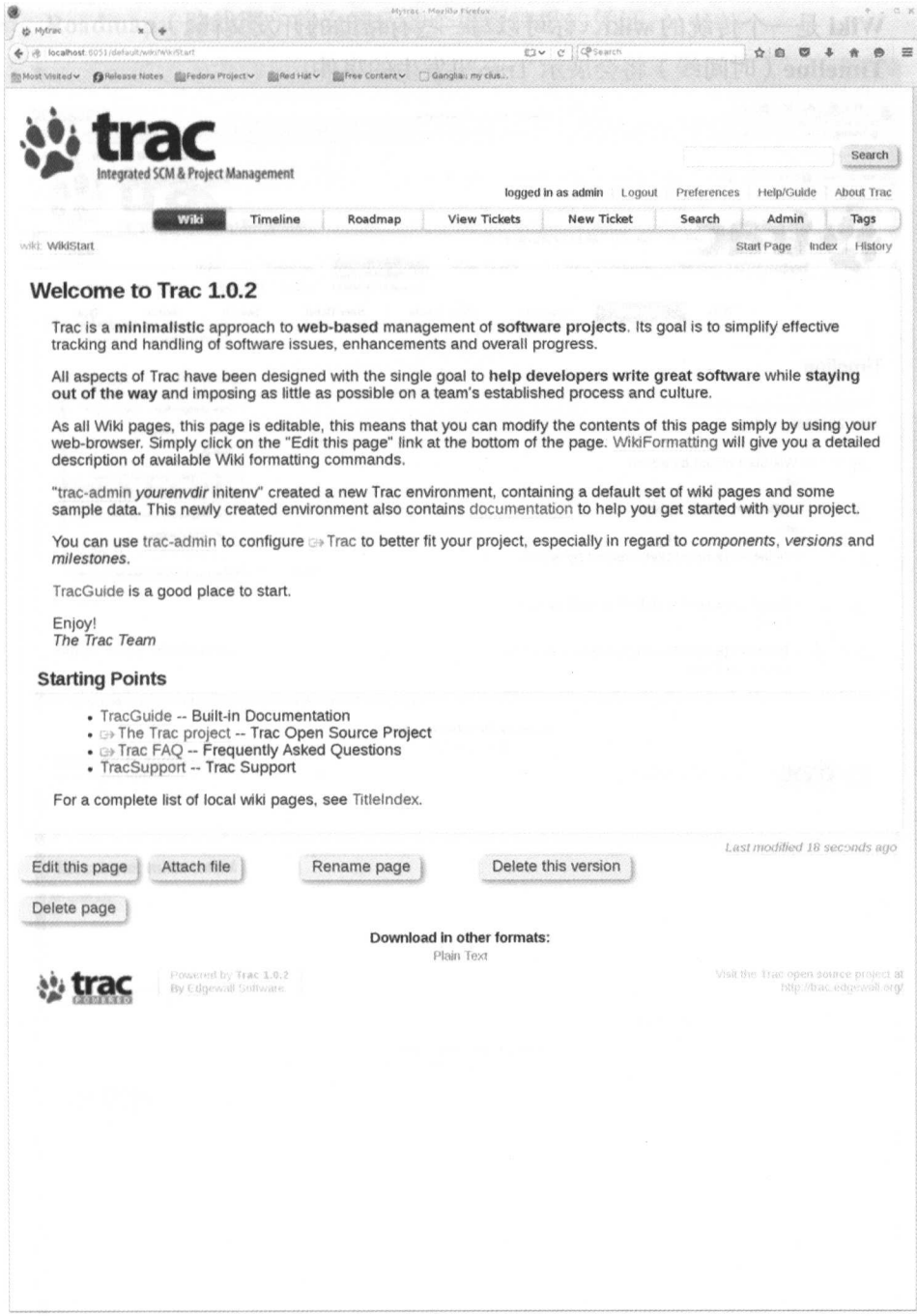
在 Docker hub 上有许多 Trac 的镜像可用，一般在 Linux 发行版的库里也是可用的。

在本书的代码包里有一个 Trac 的 Docker 文件，我们也可以用它。看一看下面的命令：

```
docker run -d -p 6051:8080 barogi/trac:1.0.2
```

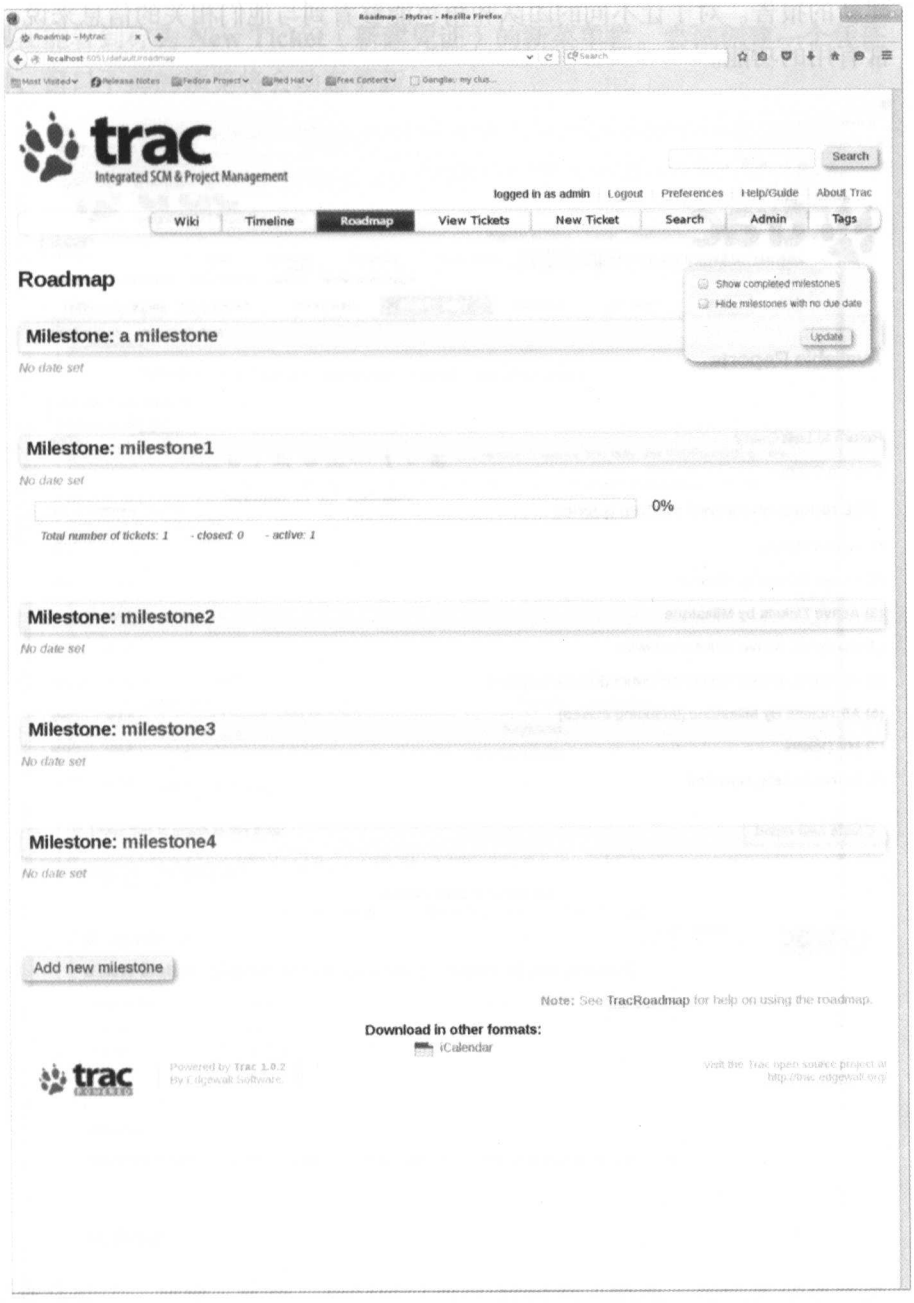
这会让 Trac 的实例运行在端口 6051 上。让我们来试一试：

- 访问 `http://localhost:6051/` 可以看到可用项目的列表。刚开始，只有默认的项目可用。
- 右边的菜单行展示了默认的基本功能，可以在下面的截图中看到：

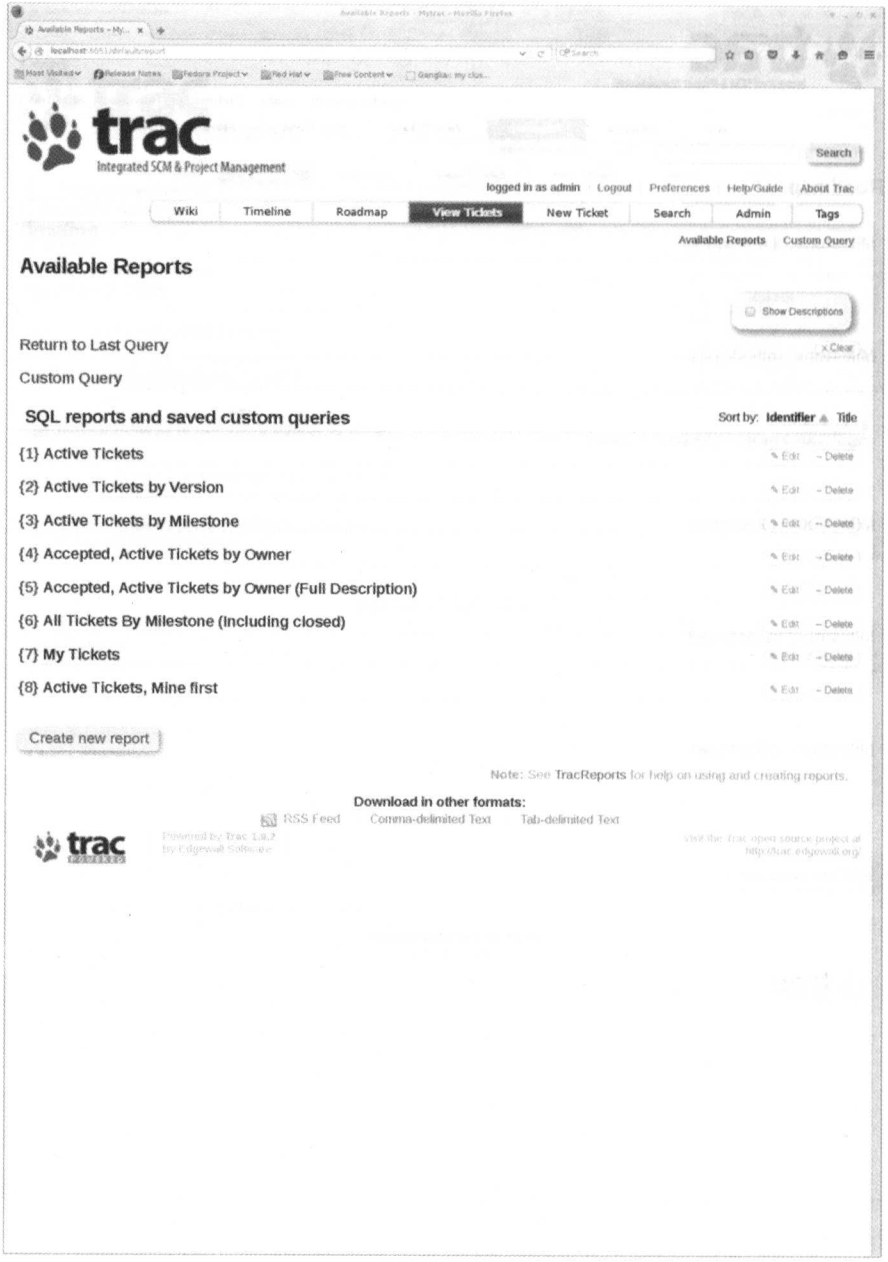

Mytrac - Mozilla Firefox
Mytrac
localhost:8051/default/wiki/WikiStart
Search
Most Visited
Release Notes
Fedora Project
Red Hat
Free Content
trac
Integrated SCM & Project Management
Search
logged in as admin
Logout
Preferences
Help/Guide
About Trac
Wiki
Timeline
Roadmap
View Tickets
New Ticket
Search
Admin
Tags
wiki: WikiStart
Start Page
Index
History
Welcome to Trac 1.0.2
Trac is a minimalistic approach to web-based management of software projects. Its goal is to simplify effective tracking and handling of software issues, enhancements and overall progress.
All aspects of Trac have been designed with the single goal to help developers write great software while staying out of the way and imposing as little as possible on a team's established process and culture.
As all Wiki pages, this page is editable, this means that you can modify the contents of this page simply by using your web-browser. Simply click on the "Edit this page" link at the bottom of the page. WikiFormatting will give you a detailed description of available Wiki formatting commands.
"trac-admin yourenvdir initenv" created a new Trac environment, containing a default set of wiki pages and some sample data. This newly created environment also contains documentation to help you get started with your project.
You can use trac-admin to configure Trac to better fit your project, especially in regard to components, versions and milestones.
TracGuide is a good place to start.
Enjoy!
The Trac Team
Starting Points
TracGuide -- Built-in Documentation
The Trac project -- Trac Open Source Project
Trac FAQ -- Frequently Asked Questions
TracSupport -- Trac Support
For a complete list of local wiki pages, see TitleIndex.
Last modified 18 seconds ago
Edit this page
Attach file
Rename page
Delete this version
Delete page
Download in other formats:
Plain Text
trac
Powered by Trac 1.0.2
By Edgewall Software
Visit the Trac open source project at
http://trac.edgewall.org/

- **Wiki** 是一个传统的 wiki，你可以写一些有帮助的开发文档。
- **Timeline**（时间线）将会展示 Trac 里发生的事件：

- **Roadmap**（路线图）将会展示各个里程碑里的凭证：



- **View Tickets（显示凭证）**，顾名思义，显示凭证。有许多可用的报告，还可以创建新的报告。对于让不同的团队和角色能够看到与他们相关的信息来说，这是非常有用的功能。



现在，试试使用用户名和密码都是 admin 来登录。

你现在能看到称为 **New Ticket（新建凭证）**的新菜单栏。尝试创建一个凭证。在你输入的同时，窗口下部的预览将会动态更新。

凭证现在处于“新建”状态。我们可以通过 **Modify（修改）**按钮来变更状态。尝试解

决凭证并添加备注。下一个状态将是“关闭”。

因为 Trac 的主要卖点之一就是 Wiki、问题跟踪器与库的紧密集成，让我们来试试这个集成的功能。编辑 wiki 首页。Trac 使用一种简单的 wiki 语法。创建一个新的标题，像下图那样使用一个等号来标记：

当我们提交变更时，问题的识别码就被激活，于是我们可以单击它来访问这个问题。

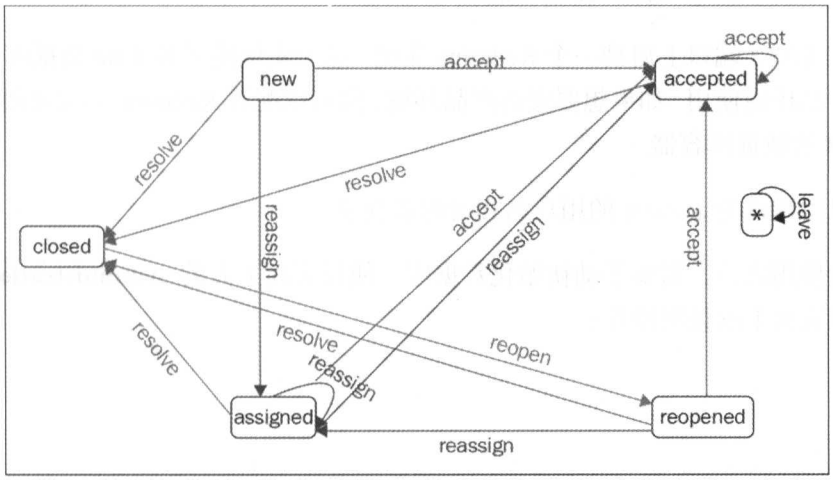
当你在代码变更的提交信息里提到一个缺陷，Trac 将会让它像在浏览器里那样可以单击。这是一个简单而方便的 wiki 功能。

Trac 的管理员界面允许自定义。你可以在 Trac 直接通过界面管理许多实体，但是不可能在这里管理 Trac 的所有配置。需要在文件系统里配置下面中的某些部分：

- 用户
- 组件
- 里程碑
- 优先级
- 决议
- 严重性
- 凭证类型

Trac 有一个可配置的状态机。全新安装的默认状态机只是比最简单的打开/关闭状态机稍微复杂一些。状态机的状态如下：

- 新建
- 已分配
- 已接受
- 已关闭
- 重新打开



Trac 发行版以 .ini 文件的形式自带一系列的工作流程样例，它们描述了工作流程的状态和改变。目前还不支持使用界面来编辑工作流程。

这是我们对 Trac 总结的评价。如果你喜欢它的简单和集成的工具，它很有吸引力。

Redmine

Redmine 是 Ruby on Rails 实现的一个流行的问题跟踪器。它的设计与 Trac 有许多相似之处。

Redmine 有两个 fork：ChiliProject 和 OpenProject。虽然 ChiliProject 已经停止了，但 OpenProject 仍然在开发中。

Redmine 是免费软件。它集成了许多工具，感觉上与 Trac 非常相似。它像我们在此研究的其他问题跟踪器一样，也是基于 web 的。

与 Trac 的开箱即用相比，Redmine 提供了一些额外的功能。下面是其中的一部分：

- web 界面可以管理多个项目。
- 可以使用甘特图和日历。

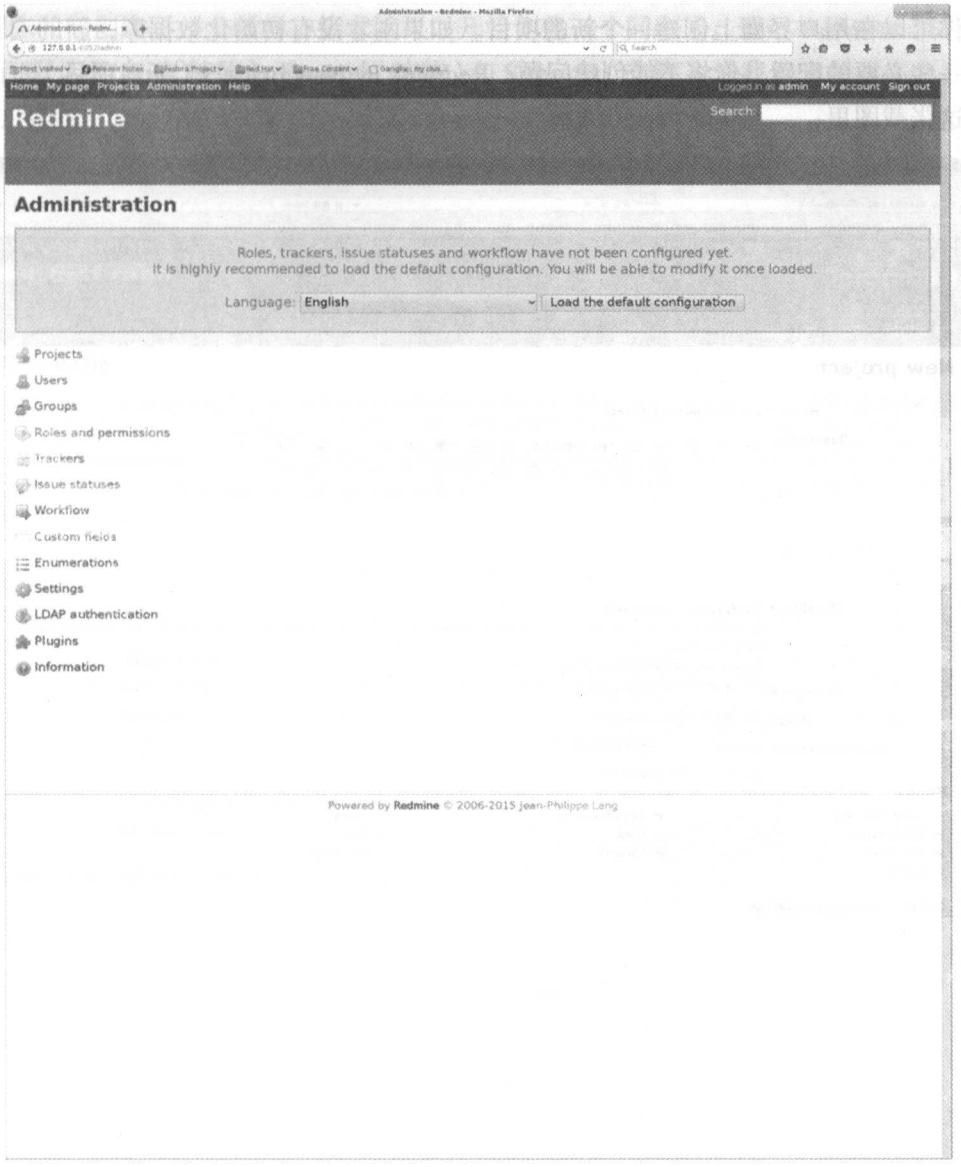
本书的代码包里有一个 Redmine 的 Docker 文件，在 Docker hub 上也有一个官方的 Redmine 镜像。使用这条命令：

```
docker run -d -p 6052:3000 redmine
```

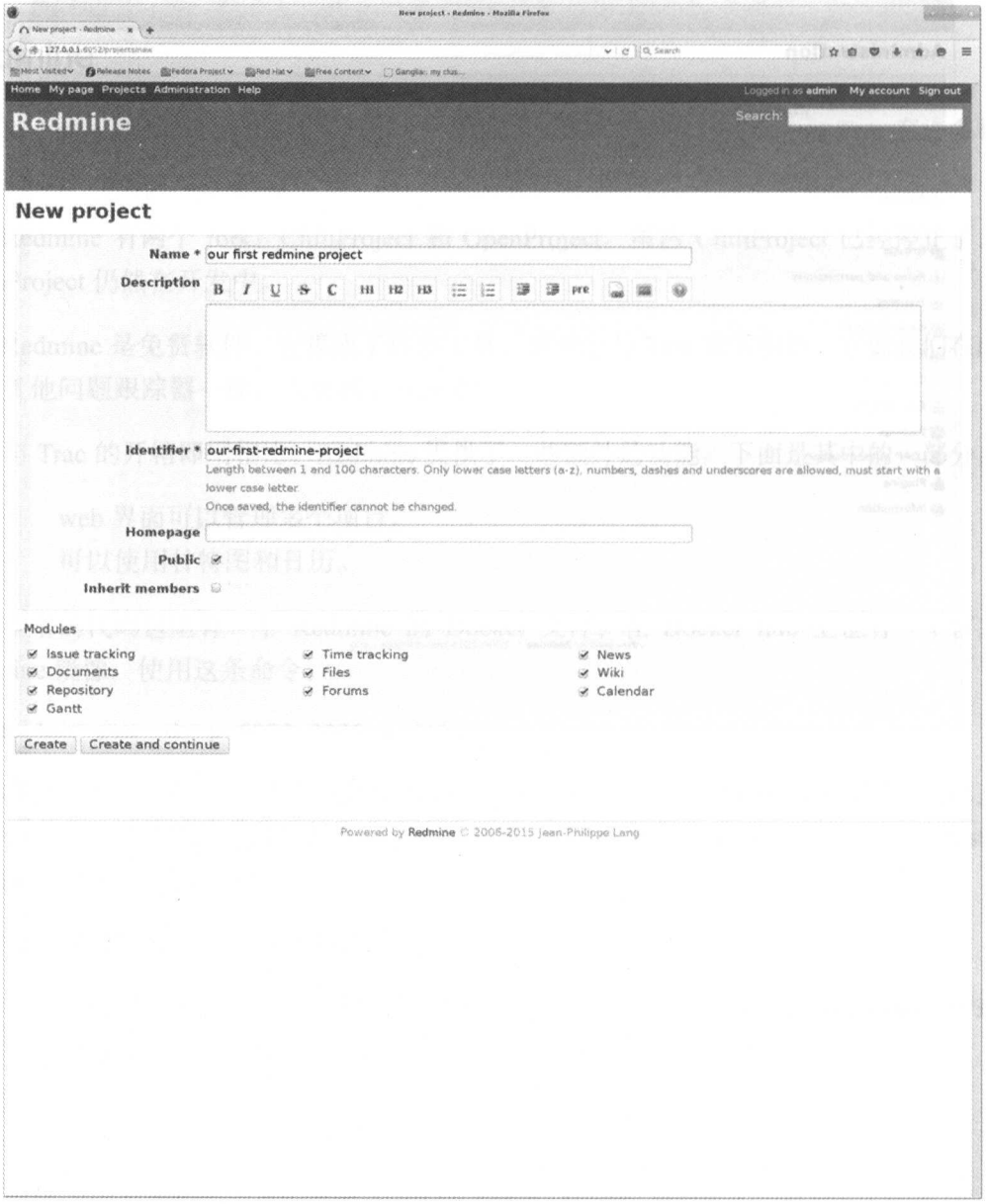
将会在 6052 端口上启动一个 Redmine 实例。这个实例使用 SQLite 数据库，所以并不适合作为产品环境使用。如果想要安装产品环境，你可以配置 Redmine 容器来使用 Postgres 或 MariaDB 的数据库容器。

你可以用默认为 admin 的用户名和密码来登录。

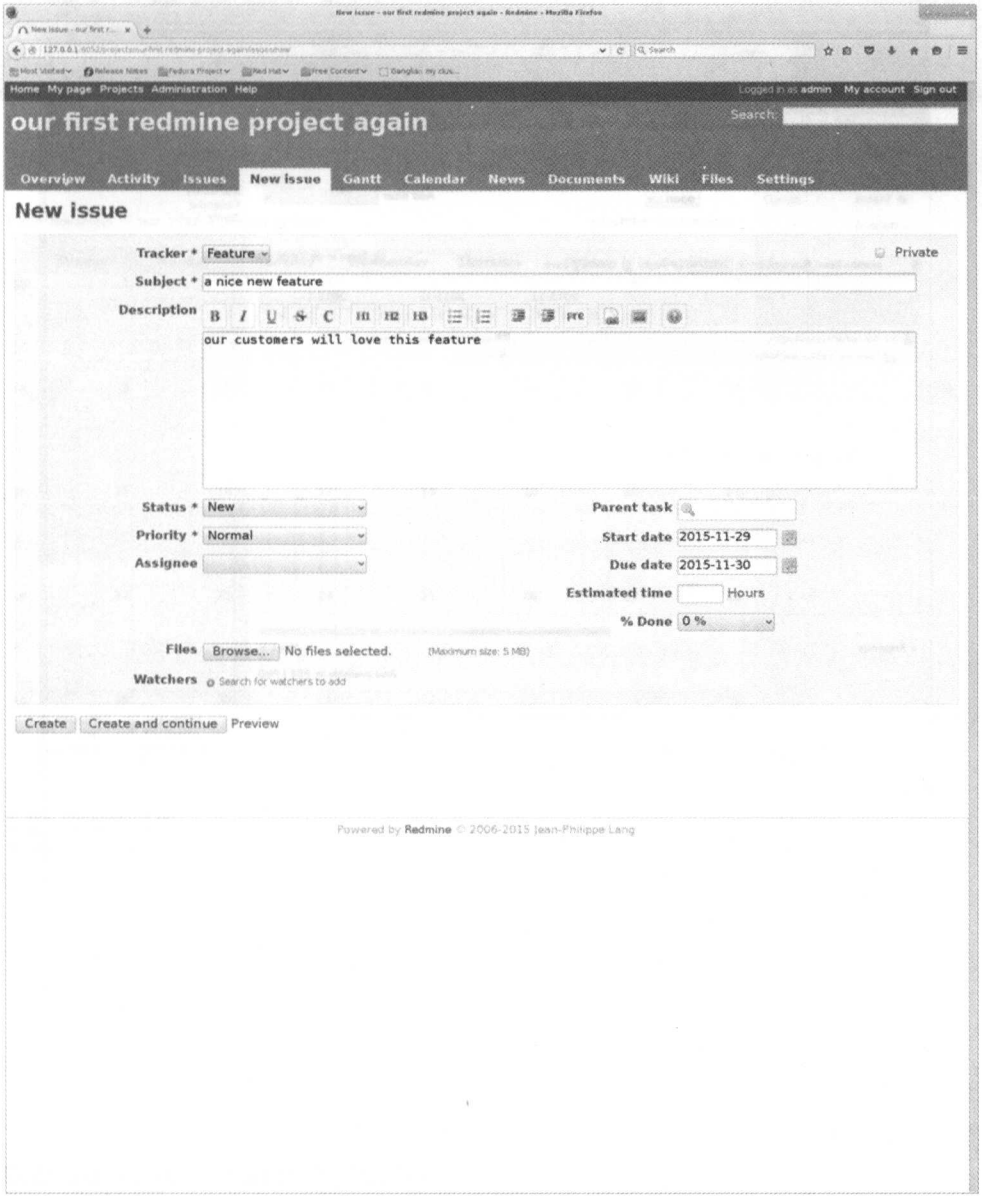
在继续使用之前，需要手动初始化数据库。使用页面上方的 **Administration（管理）**链接。按照页面上的说明操作：


Administration - Redmine - Mozilla Firefox
Home My page Projects Administration Help
Logged in as admin My account Sign out
Redmine
Search:
Administration
Roles, trackers, issue statuses and workflow have not been configured yet.
It is highly recommended to load the default configuration. You will be able to modify it once loaded.
Language: English
Load the default configuration
Projects
Users
Groups
Roles and permissions
Trackers
Issue statuses
Workflow
Custom fields
Enumerations
Settings
LDAP authentication
Plugins
Information
Powered by Redmine © 2006-2015 Jean-Philippe Lang

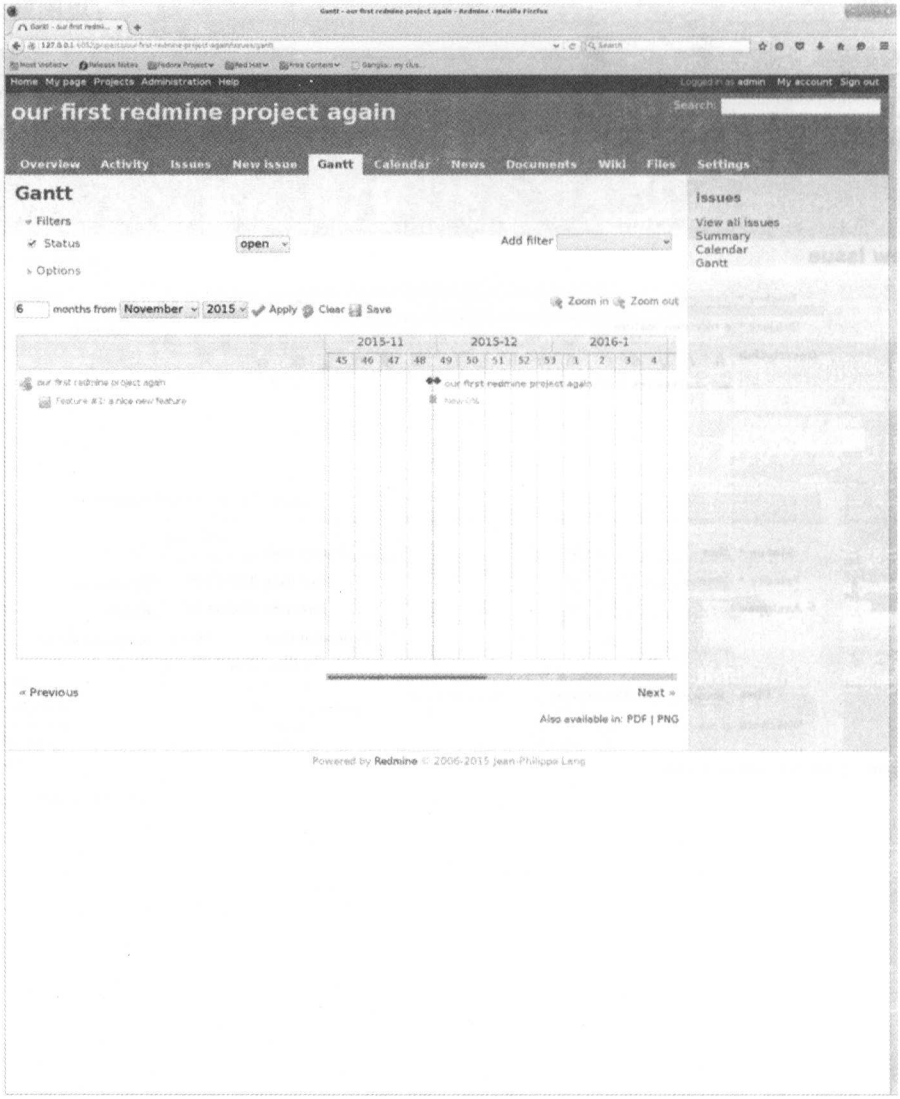
你可以在用户界面上创建一个新的项目。如果刚才没有初始化数据库，新的项目就会缺失一些必要的配置，你将不能创建问题。Redmine 的 **New project**（**新建项目**）页面就在下面这张截图里：



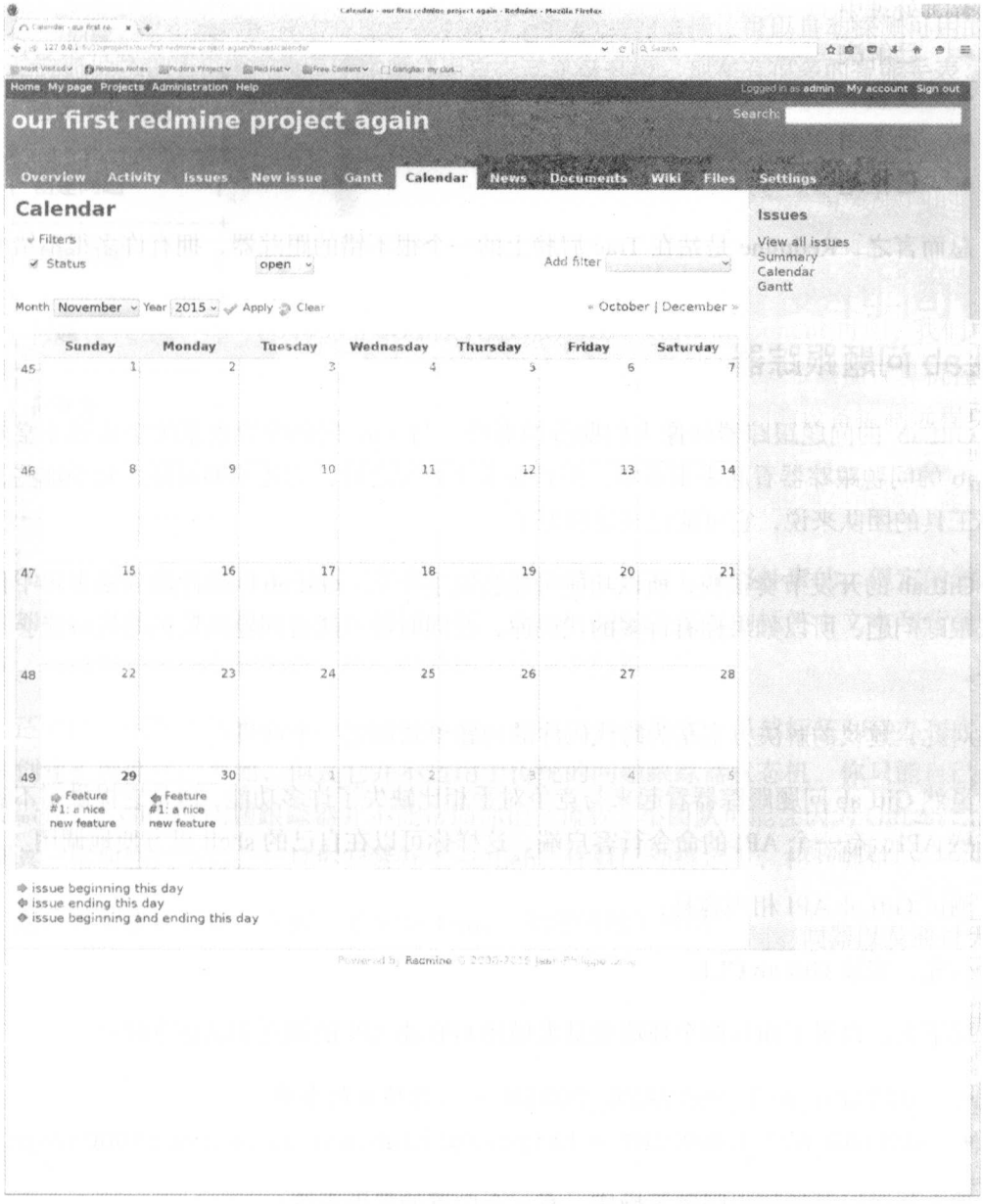
现在我们创建了一个项目，接着就可以开始创建一个问题了。通过 Redmine 选择问题的类型：默认有 **Bug**（**缺陷**）、**Feature**（**功能**）和 **Support**（**支持**）。让我们来创建一个 feature 类型的问题：



在其他众多的功能之中，还有 **Gantt**（甘特）视图和 **Calendar**（日历）视图。这就是甘特图：



然后是日历图：



Redmine 里问题的默认状态机如下：

- 新建
- 处理中
- 已解决
- 反馈
- 已关闭
- 已拒绝

总而言之，Redmine 是站在 Trac 肩膀上的一个很不错的跟踪器，拥有许多很不错的功能。

GitLab 问题跟踪器

GitLab 的问题跟踪器就像人们期待的那样，与 Git 代码库管理系统集成得非常好。GitLab 的问题跟踪器看起来很不错，但是在本文撰写之时，它还不够灵活。对于那些喜欢简单工具的团队来说，它可能已经足够好了。

GitLab 的开发节奏很快，所以功能可能会发生变化。GitLab 问题跟踪器根据每个代码库来跟踪问题，所以如果你有许多的代码库，在同时展示所有问题概览的时候可能会比较难办。

对此，建议的解决方案是为跨代码库的问题单独创建一个项目。

虽然 GitLab 问题跟踪器看起来与竞争对手相比缺失了许多功能，但是它提供了不错的可扩展 API，有一个 API 的命令行客户端，这样你可以在自己的 shell 里方便地调用。

测试 GitLab API 相当容易：

首先，安装 GitLab CLI。

接下来，设置下面这两个环境变量来描述 GitLab API 的端点和认证令牌：

- `GITLAB_API_PRIVATE_TOKEN = <你项目的令牌>`
- `GITLAB_API_ENDPOINT = http://gitlab.matangle.com:50003/api/v3`

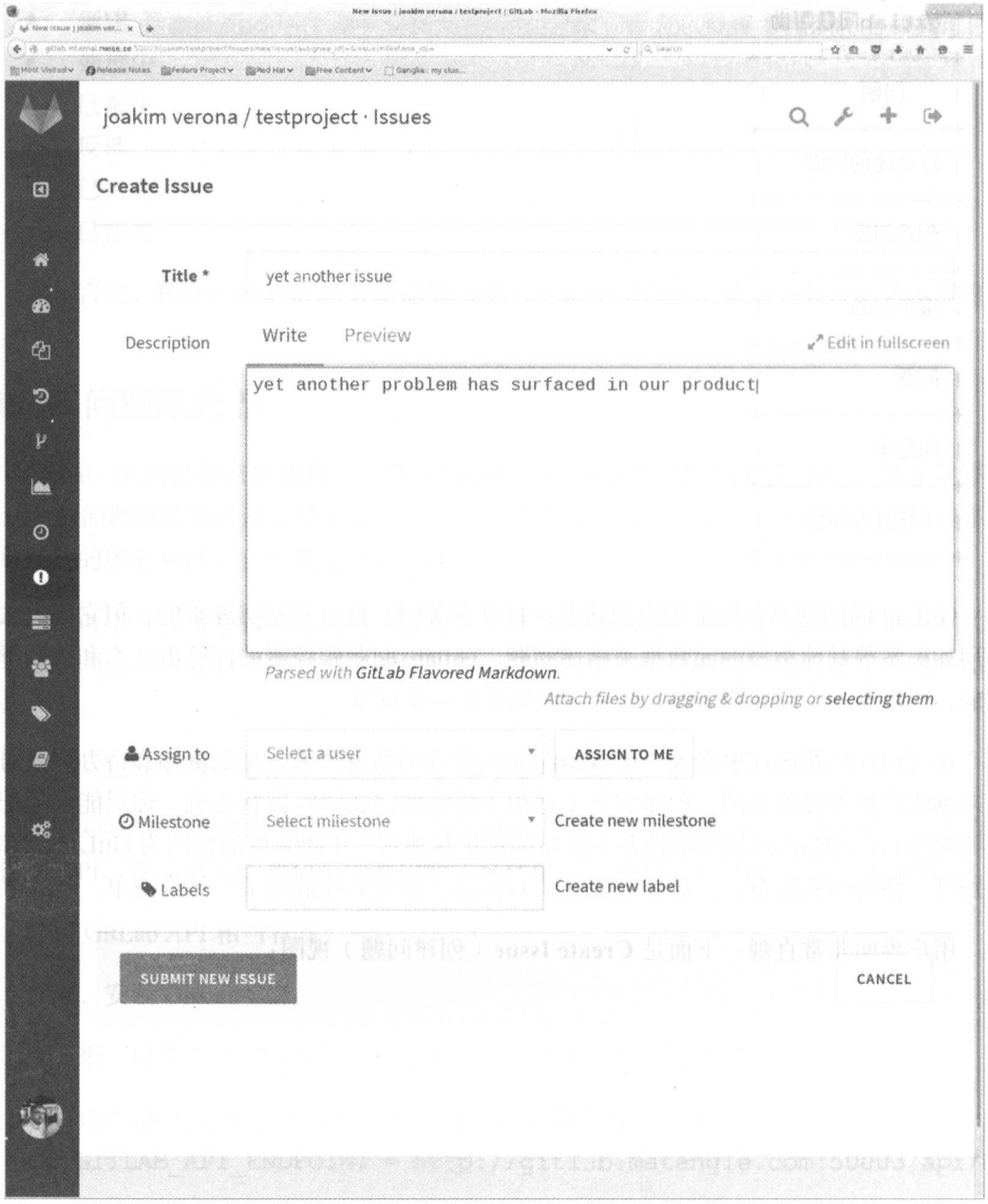
现在你可以开始列出问题等操作。有一个内置的帮助系统：

```
gitlab问题帮助
+--------------+
|    问题      |
+--------------+
| 已关闭的问题  |
+--------------+
| 创建问题      |
+--------------+
| 编辑问题      |
+--------------+
| 问题          |
+--------------+
| 问题集        |
+--------------+
| 重新打开问题  |
+--------------+
```

GitLab 的问题只有两个约定的状态：打开和关闭。这可真是够朴素的，但它的做法是使用如标签等其他类型的元数据来描述问题。Gitlab 问题跟踪器的标签由文本和背景颜色组成，可以与一个问题相关联，还可以关联不止一个标签。

还可以在问题描述里嵌入一个 Markdown 格式的待办列表。虽然标签和待办列表可以用来创建扩展性强的视图，但是它并不适用于传统的问题跟踪器状态机。你只能自己记住需要做什么：GitLab 问题跟踪器并不能帮助你记住流程。小团队可能会因为 GitLab 跟踪器的别具一格而感到振奋，一旦你安装好了 GitLab，你就已经搞定了，相当简单。

用户界面非常直观。下面是 **Create Issue**（创建问题）视图：


joakim verona / testproject · Issues
Create Issue
Title *
yet another issue
Description
Write
Preview
Edit in fullscreen
yet another problem has surfaced in our product
Parsed with GitLab Flavored Markdown.
Attach files by dragging & dropping or selecting them.
Assign to
Select a user
ASSIGN TO ME
Milestone
Select milestone
Create new milestone
Labels
Create new label
SUBMIT NEW ISSUE
CANCEL

下面是一些界面的功能：

- **分配给**：可以被分配的人相当于项目的成员。
- **标签**：你可以用一套默认的标签，或是自己创建，或是混合使用。这里列出了默认的标签：
  - 缺陷
  - 已确认
  - 非常严重
  - 讨论
  - 文档
  - 功能增强
  - 建议
  - 支持

下面有一些可以用于问题的属性：

- **里程碑**：你可以把一些问题归拢为里程碑。里程碑以时间线的方式描述了问题应该何时被关闭。
- **GitLab 风格的 Markdown 支持**：GitLab 支持 Markdown 和一些其他的标记，在 GitLab 里连接不同的实体更加容易。
- **添付文件**。

在复杂的 Bugzilla、Trac 和 Redmine 界面之后，GitLab 问题跟踪器简单直接的方式可以令人耳目一新！

Jira

Jira 是一个灵活的缺陷跟踪器，由 Atlassian 公司用 Java 编写。它是我们评估中唯一的收费问题跟踪器。

虽然我喜欢 **FLOSS（Free/Libre/Open Source Software，免费/自由/开源软件）**，但是 Jira 非常适用，对于 15 人以内的小团队是免费的（译者注：现在可不免费了，具体价格请参考 `https://www.atlassian.com/software/jira/pricing?tab=host-on-your-server`）。与 Trac 和 GitLab 不同，像 wiki 和代码库展示器那样的不同模块是独立的产品。如果需要一个 wiki，你就得单独部署，Atlassian 自营的 Confluence Wiki 可

以与之顺畅集成。Atlassian 也为代码库的集成和浏览提供了一个称为 **Fisheye** 的代码浏览器。

对于小团队来说，Jira 的许可证很便宜，但是随着人数增加会变的很贵。Jira 默认的工作流程很像 Bugzilla：

- 打开
- 处理中
- 已解决
- 已关闭
- 重新打开

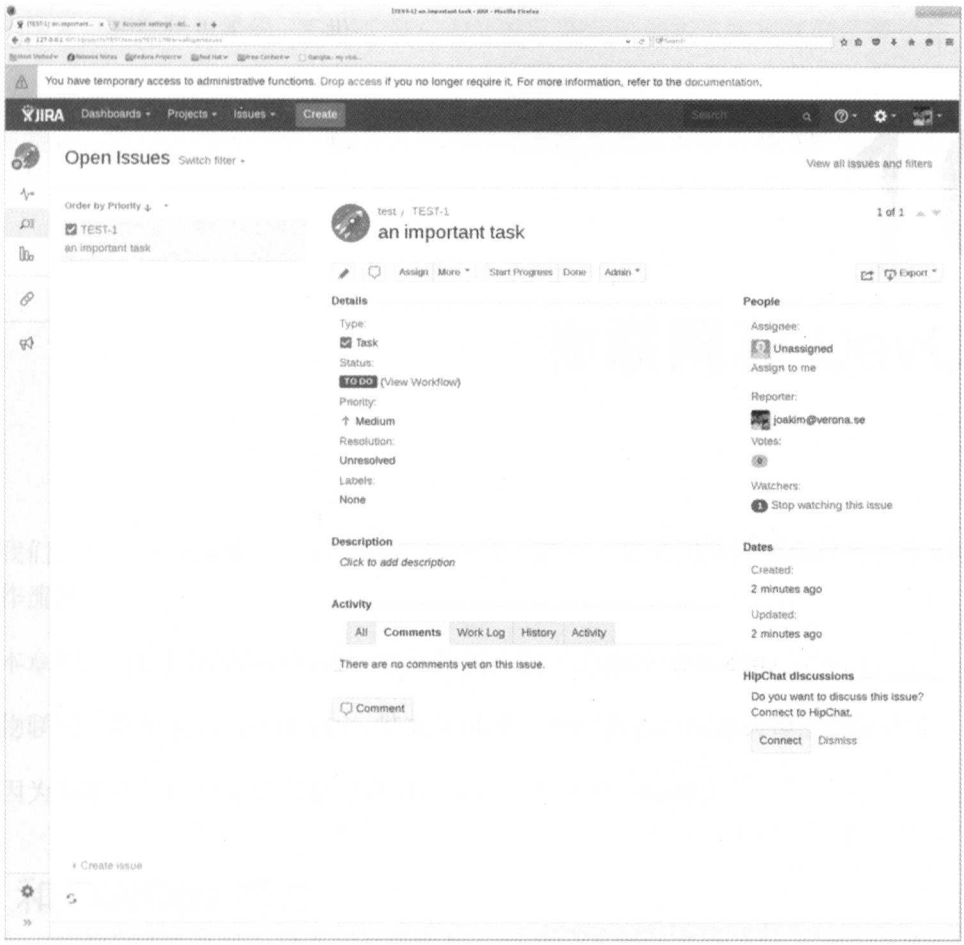
Docker Hub 有几个可用的镜像，`cptactionhank/atlassian-jira` 是其中之一——但是请记住，Jira 不是免费软件。

现在我们来试用 Jira：

```
docker run -p 6053:8080 cptactionhank/atlassian-jira:latest
```

你可以通过端口 6053 访问界面。试用 Jira 有点儿复杂，因为需要事先创建一个 Atlassian 的账户。完成之后用自己的账户登录。

Jira 有一个简单的入门教程，你可以创建一个项目并在里面创建一个问题。如果你照办了，结果视图看起来会像这样：



Jira 提供了许多功能，不少功能都可以通过内置的应用商店以插件的方式使用。如果你能承受 Jira 的费用，那么它可能是我们探索过的问题跟踪器里看上去最高大上的那个。它的配置和管理也相当复杂。

总结

本章中，我们探索了许多不同的系统，可以实现在企业里支持问题跟踪流程。照样有许多可选的方案，尤其是这个领域，都是天天打交道的工具。

下一章将会介绍稍微更有深度的一些东西：DevOps 和物联网。

10

物联网和 DevOps

我们在上一章中探索了许多不同工具中的一部分，如可用的问题跟踪器，帮助我们管理工作流程。

本章的内容比较有前瞻性：DevOps 如何在新兴的物联网领域帮助我们。

物联网，简称 IoT，给 DevOps 带来了挑战。我们将会介绍这些挑战都是什么。

因为 IoT 这个词的定义不是很清晰，让我们先了解一些背景。

IoT 和 DevOps 简介

物联网这个词来源于 20 世纪 90 年代末期，据称由英国的企业家 Kevin Ashton 提出，当时他正在用 RFID 技术。Kevin 在宝洁公司工作时开始对使用 RFID 来管理供应链产生了兴趣。

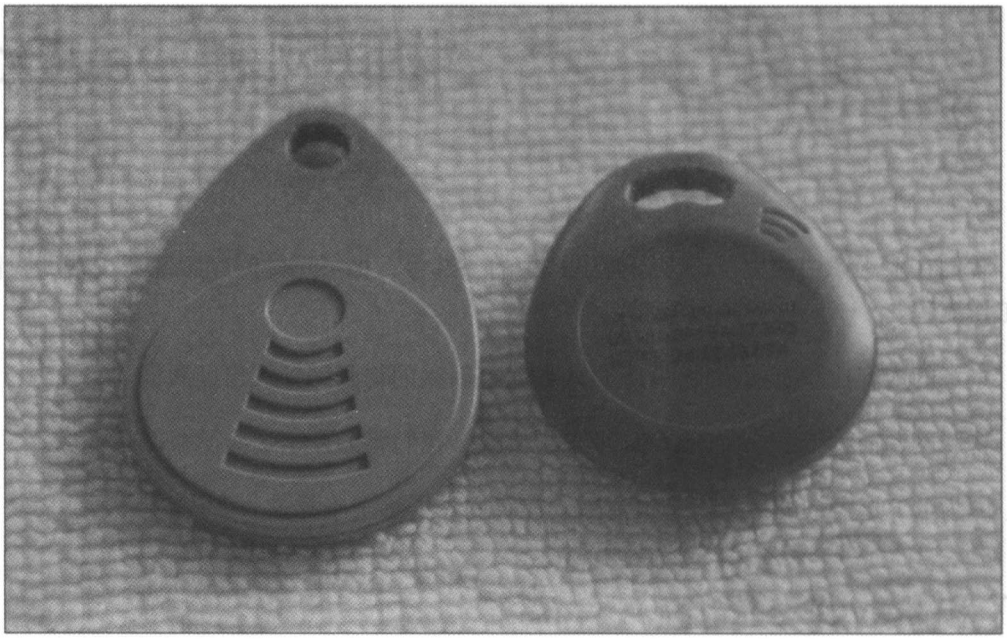
RFID，全称为**射频 ID（Radio Frequency ID）**，举个例子，是你钥匙链上用来开门的小标签背后的技术。在这个场景下，RFID 标签间接地成为物联网的例子。当然，RFID 标签并不局限于开门的功能，外形也不局限于钥匙链上的标签。

RFID 标签包含一个大约 2 平方微米的小芯片和一个线圈。放置在读取器附近时，线圈通过感应带电，芯片获得了足够多的能量来发送一个唯一的标识符到读取器的硬件。反过来读取器发送标签字符串到服务器，服务器决定如何响应，比如要不要打开关联到读取器

的锁。服务器可能连接到互联网，根据不断变化的开门权限，远程添加或删除 RFID 标签相关的鉴定器。由好几个不同的系统共同协作来达到想要的结果。

其他有趣的物联网技术还包括通过摄像头扫描二维码来为系统提供信息。新的低功耗蓝牙技术提供了智能有源传感器。这样的传感器可以用锂电池续航一年。

下面是两个操作锁的 RFID 的标签：



物联网这个词非常模糊。什么时候这是个“物”，什么时候是计算机？或者它是不是两者兼有？

让我们以小型计算机，例如树莓派为例，它是一个**芯片系统（SoC）**，挂载在一个信用卡大小的板子上。虽然小，但是它仍然是一个完整的计算机，足以强大到运行 Java 应用服务器、web 服务器和 Puppet 代理。

与传统的计算机相比，物联网设备有如下几种限制。通常，它们都是嵌入式设备，位置受到限制，放在很难访问到的地方。从 DevOps 的视角来说这点似乎是定义物联网的特征：设备可能会受到不同方面的制约。我们不能使用服务器或者桌面计算机上的所有技术。举几个例子，这些限制可能是有限的内存、处理能力或者访问能力。

下面是一些物联网设备的种类，你可能已经在什么地方见过或者将要见到：

- **智能手表**：今天，一个智能手表可以有蓝牙和无线连接，可以自动探测可用的更新。它可以下载新的固件并通过用户交互自己升级。现在有很多的智能手表，从粗糙的 Pebble 到 Android Wear 和 Apple Watch。带有移动网络连接的智能手表也已经出现，最终它们会得到普及。
- **键盘**：键盘也可以有升级的固件。从某种意义上说，键盘是另一个物联网的好例子。它有很多传感器。可以运行提供传感器输出到连接互联网的更强大的机器。现在有完全可编程且固件开放的键盘，比如 Ergodox，它的类 Arduino 板接口连接到键盘矩阵（`http://ergodox.org/`）。
- **家庭自动化系统**：这些都连接到互联网并可以通过智能手机或者桌面计算机控制。Telldus TellStick 就是这样的设备，可以让你通过连接互联网的设备控制远程功率继电器。类似这样的系统有很多。
- **无线监控摄像机**：这些可以用智能手机界面监控。针对不同的细分市场，有不同的供应商，比如 Axis Communications。
- **生物传感器**：这样的传感器包括健身传感器，比如脉搏计、体重秤和放在身体上的加速计。Withings 体重秤测量生理数据并将其上传到服务器，你可以通过网站来获取这些数据。智能手表和手机中的加速计可以跟踪你的运动状况。
- **基于蓝牙的钥匙寻找器**：如果你丢掉了钥匙，可以从智能手机上激活它。
- **车载电脑**：它可以做所有的事情，从多媒体播放到管理车中物理控制系统的导航，比如锁、窗户和门。
- **音频和视频系统**：这包括娱乐系统，比如网络音乐播放器和视频流硬件，比如 Google Chromecast 和 Apple TV。
- **智能手机**：无处不在的智能手机真的是连接 3G 或者 4G 调制解调器以及通过无线连到互联网的小计算机。
- **嵌入在存储卡具有 Wi-Fi 功能的计算机**：这让单反相机变成一个有 Wi-Fi 的相机，可以自动将图片上传到服务器。
- **家里或者办公室的网络路由器**：这些都是事实上的小服务器，经常可以从 ISP 端远程升级。
- **网络打印机**：这都是现在非常智能并且可以和云服务器交互以方便从不同的设备打印。

当然这个列表还可以继续，所有的这些设备都已经出现并且很多家庭都已经使用了。

有趣的是，从技术的角度看，这里提到的很多设备都很相似，即便它们用于完全不同的目的。多数的智能手机都使用了不同的 ARM 处理器架构，可以和一些外围芯片的变体跨厂商联合授权。比如，MIPS 处理器在路由器硬件提供商里很流行，Atmel RISC 在嵌入式硬件实现者里使用很广。

开发者和爱好者可以使用基本的 ARM 芯片去做原型设计。树莓派基于 ARM 并且可以用来为专业的设备做原型设计。同样，Arduino 可以为 Atmel 架构的设备做硬件的原型设计。

这使得物联网的创新可能成为完美风暴：

- 该设备价格便宜，即使数量小也容易获得。
- 设备开发和获得原型平台都比较简单。
- 开发环境相似，学习成本低。很多时候都使用 GNU 编译器集合或者 GCC。
- 已经有大量的论坛、邮件列表和文档等的支持。

像树莓派这样强大的设备，我们可以使用在服务器上一样的方法和实践。树莓派设备可以作为服务器，只是性能不如传统的服务器。对于物联网设备来说，无代理的部署系统比需要代理的系统更适合。

更小一点的设备，比如 Arduino 中使用的 Atmel 嵌入式处理器，受到的限制更多。通常情况下，当特殊的启动加载程序代码运行时，你可以编译新的固件并在重新启动时将它们部署到该设备。之后，设备通过 USB 连接到主机。

在开发的过程中，可以通过连接单独的设备复位原始设备并把它变为加载模式来自动化上传固件。在开发过程中可能没有太大问题，但是在实际的部署场景下因为成本的缘故这样做性价比不高。这些都是 DevOps 在物联网领域可能遇到的问题。在开发环境中，我们可能可以或多或少使用我们过去开发服务器应用的方法，也许需要额外的硬件。尽管从保证质量的角度来看，部署和测试使用不同的硬件存在一定的风险。

从市场的角度看物联网的未来

根据 Gartner 的研究，到 2016 年末，连接到互联网的设备大约会有 64 亿：比 2015 年增加了 30%。再往前到 2020 年底，将大约会有 210 亿连接到互联网的设备。消费者市场的

设备将会是最多的，而企业的支出将会是最大的。

下一代的无线网络，姑且称为 5G 移动网络，估计会在 2020 年进入市场，离本书撰写的时候已没有几年。即将到来的移动网络拥有适于物联网设备的能力并且远远领先于今天的 4G 网络。

一些新的移动网络标准对应的能力包括：

- 连接比今天的网络更多设备的能力。这将使传感器网络部署成千上万的传感器成为可能。
- 为成千上万的用户提供几十兆每秒的数据速率。在办公室环境中，每个节点都有千兆每秒的容量。
- 低延迟，实现实时交互应用。

不论 5G 的提议在实际中如何实现，我们可以安全地假设移动网络会进化得更快，更多的设备能直接连到互联网。

让我们看看业界的巨头都是怎么说的：

- 从瑞典移动网络硬件的领导者爱立信开始。下文引用自爱立信物联网主页：

  “爱立信和物联网

  超过 40%的全球移动网络流量运行在爱立信的网络上。我们是业界最具创新的公司，拥有超过 35 000 件授权的专利，我们正在通过为业务降低门槛来引领物联网的变革，通过现代通信技术，通过打破不同行业间的壁垒，通过连接企业、人和社会创建新的解决方案。展望未来，我们看到了两个重大机遇领域：

  产业转型：物联网成为一个又一个部门的关键因素，实现新型服务和应用，改变商业模式，创造新的交易市场。

  不断发展的运营商角色：在新的物联网的模式下，我们可以看到，运营商可以承担三种切实可行的不同角色。运营商选择哪些角色追求取决于历史、抱负、市场的先决条件和它们的未来前景。”

爱立信还贡献了一个漂亮物理网概念的字典，称为“漫画书”：

```
http://www.alexandra.dk/uk/services/Publications/
Documents/IoT_Comic_Book.pdf
```

- 网络巨人思科预测到 2020 年将有超过 500 亿的物联网设备连接到互联网。

  思科倾向于使用“万物物联（Internet of Everything）”而不是物联网这个词。这家公司设想了很多前面没有提到过的有趣应用。其中一些如下：

  - 智能垃圾桶，其嵌入的传感器允许远程感知垃圾桶是不是满了。从而可以改善废物管理的组织工作。
  - 停车计费器可以按需改变费率。
  - 如果用户生病可以发出通知的衣服。

  所有的这些系统都能一起连接和管理。

- IBM 的预测较为保守，到 2020 年底将有 260 亿的设备连接到互联网。他们提供了一些已经部署或在开发中的应用的例子：
  - 通过汽车制造商广泛采用传感器网络部署来完善物流。
  - 医院医疗保健情况的更智能的后勤管理。
  - 更聪明的物流和对火车旅行更精准的预测。

不同的市场主体对未来的预测稍微有些不同。不论是哪种速率，可以清楚地看到在未来几年物联网将呈指数级增长。很多新的激动人心的应用将呈现在大家眼前。很明显，许多领域将受到这种增长的影响，DevOps 领域也不能幸免。

机器到机器的通信

很多的物联网设备将主要连接到其他机器。从规模的角度比较，让我们假设每个人都有一台智能手机。总的设备数大约是 50 亿。物联网最终将拥有至少 10 倍以上的设备——500 亿设备。对于这件事情发生的时间大家有些分歧，但是最终我们将会到达那里。

这种增长的一个驱动力是机器到机器的通信。

工厂已经逐渐向更大程度的自动化转移，这种趋势会随着越来越多的方案变得可用而

增加。工厂内部物流通过广泛部署的传感器网络效率大幅提升，通过大数据分析持续改进流程。

现代化汽车为每个可能的功能使用处理器，从灯光、仪表盘到车窗升降。下一步将连接汽车到互联网，最终自动驾驶汽车会将所有传感器数据传送到集中协调服务器。

许多形式的旅行可以受益于物联网。

物联网的部署影响软件架构

物联网由很多设备组成，它们也许没有使用相同的固件版本。升级的时间可能是分散的，因为有时候硬件不是物理可用的状态。这使得接口层面的兼容性变得很重要。因为小的网络传感器会受制于内存或者处理器，版本二进制协议或者简单 REST 协议可能更受欢迎。版本协议在允许具有不同硬件版本的物品在不同版本的端点进行通信时很有用。

大规模的传感器部署可以从交互少的通信协议和分层消息队列架构异步处理事件中获益。

物联网部署的安全性

安全是一个很困难的主题，一大堆设备连接到互联网而不是私有网络让情况变得更加困难。许多消费级硬件设备，如路由器，有旨在用来升级的接口，但这很容易被破解。一个合法的设备从而变成了一个后门。增加物理接口的同时也增加了可能的攻击数量。

也许你能认出下面这些部署的反模式：

- 一个开发者在代码中保留了能让他或者她以后可以提交将要在服务器应用上下文执行的代码。这个想法的打算是作为开发人员，你不知道什么样的修改是必要的，当准备部署变更时，操作人员是否准备就绪。那么，为什么不在代码中留一个后门，这样我们可以在需要的时候直接部署代码？当然，这样有很多问题。开发人员觉得平常协商好的部署过程不够有效，最终结果是黑客也可以很容易发现如何使用这个后门。这种反模式是比我们想象的更为常见，而唯一的正确补救方法是代码审查。

给黑客留门绝对不是什么好事情，举个例子，如果自动驾驶汽车或者热力工厂的后门被利用，后果会是多么不堪设想。

- 诸如SQL注入那样的恶意攻击，大多数时候是因为开发人员没有注意到这个问题。

  解决的办法是了解这些问题的相关知识，并且编码时就需要考虑如何避免这些问题发生。

好啦，但是 DevOps 和物联网有什么关系？

让我们后退一步。迄今为止，我们已经讨论了物联网的基础，基本上是寻常的互联网加上我们无法想象的节点数。我们也看到，在未来的几年中，能以各种形式联网的设备数量将继续呈指数增长。这一增长将是因特网的机器对机器部分。

但是，对于更加关注快速交付的 DevOps，真的适合关键嵌入式设备的大型网络吗？

经典的反例是 DevOps 在核设施或者在诸如心脏起搏器的医疗器械中。但是单纯地更快发布不是 DevOps 的核心理念。它是通过将不同学科的人紧密联系在一起工作，更快、更准确地发布。

这意味着让类产品环境的测试环境贴近开发者，同时大家的合作更加紧密。

这么说的话，看起来 DevOps 可以用在保守的传统行业。

当然，不能低估面临的挑战：

- 嵌入式设备的生命周期比传统的客户端——服务器计算机要长。消费者不能期望在每个产品周期都升级。同样，工业设备部署的地方更换起来可能很昂贵。
- 相比桌面计算机，物联网设备失败的模式更多。这让测试变得更加困难。
- 在工业部门和企业部门，可追溯性和可审计性是很重要的。这和在服务器上部署是一样的，但物联网端点比服务器更多。
- 传统的 DevOps 可以将很小的变更部署到用户的一个子集。如果修改不工作，我们可以修复并重新部署。如果对我们一个已知的用户群来说网页渲染很糟糕，并且这个问题可以快速修复，潜在的风险就很小。另一方面，如果一个单的物联网

设备控制的物体，例如一道门或一个工业机器人出现故障时，造成的后果可能是灾难性的。

物联网领域对于 DevOps 来说挑战很大，但是换种方式不见得会更好。DevOps 也是一个工具箱，你需要思考从中找挑出的工具是否能正确应对当前工作。

我们仍然可以使用许多 DevOps 工具箱中的工具，只需要确保我们在做正确的事情，而不只是在不理解问题的前提下实现想法。

下面是一些建议：

- 只要你在测试实验室中，失败和快速周转是可以的。
- 确保你的测试实验室和产品环境接近。
- 在实验室不要只使用最新版本，也要兼容旧版本。

DevOps 的物联网设备动手实验室

到目前为止，我们大多讨论了 DevOps 的抽象意义、物联网及其未来。

为了得到动手环节的灵感，让我们来制作一个简单的物联网设备，它可以连接到 Jenkins 服务器并且显示出构建的状态。通过这种方式，将我们尝试的物联网设备和 DevOps 结合起来！

在构建失败的情况下，将闪烁的 LED 作为状态显示。这个项目很简单，但是聪明的读者可以以此为基础扩展项目。为本次练习挑选的物联网设备比较灵活，可以实现比 LED 闪烁更多的功能。

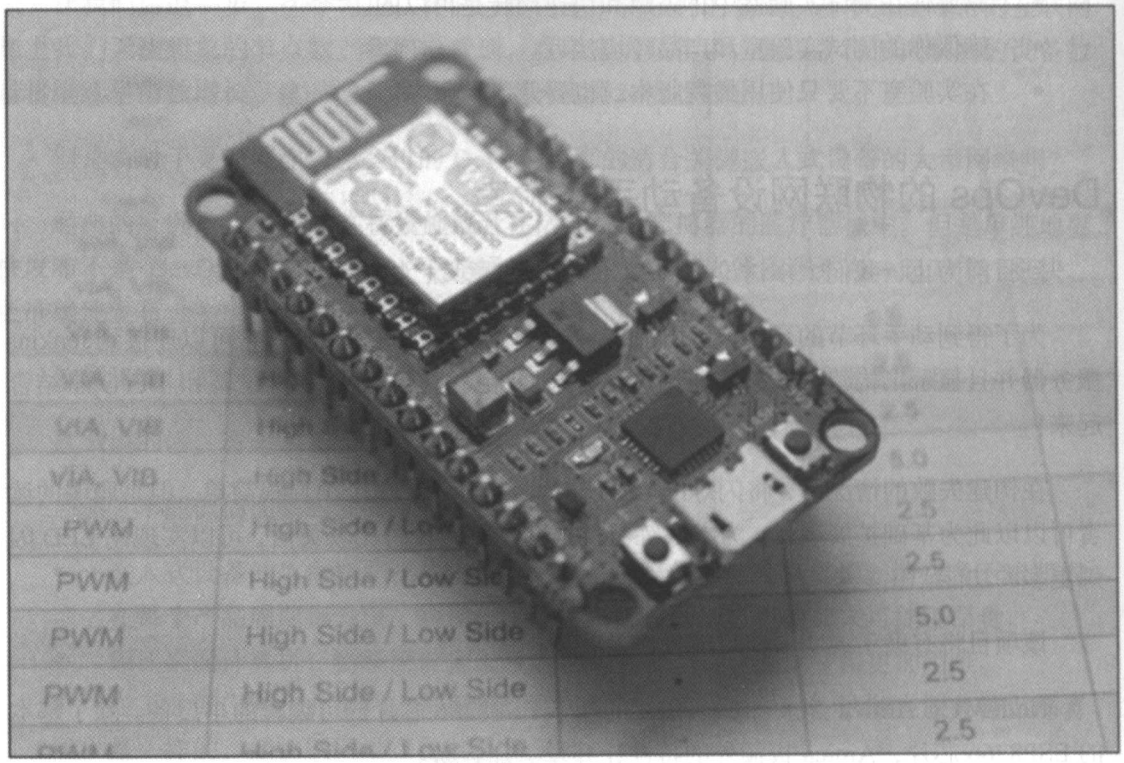
该项目将有助于说明一些可能性以及物联网的挑战。

NodeMCU Amica 是来自 Espressif 的基于 ESP8266 芯片的可编程的小设备。除了基本的 ESP8266 芯片，Amica 板额外的特性让开发更加容易。

下面是一些设计的规格：

- Tensilica Xtensa LX106 是一个 32 位的 RISC CPU，运行频率为 80MHz。
- 它的 Wi-Fi 芯片允许它连接到我们的网络和 Jenkins 服务器。

- NodeMCU Amica 板有一个 USB 接口可以给固件编程并连接到电源适配器。ESP8266 芯片需要一个 USB 到串口的适配器去连接 USB 接口，它由 NodeMCU 板提供。
- 板子有几个输入/输出的端口，可以连接到某些硬件上来可视化构建的状态。开始我们会做得比较简单，只使用连接到设备上某个端口的板载的 LED。
- NodeMCU 自带的固件可以通过 Lua 语言来编程。Lua 是一种高级语言，可以快速实现原型。顺便提一句，它在游戏编程领域也很流行，也可以从另一方面说明 Lua 的高效。
- 考虑到它提供的这么多功能，这个设备相当便宜：



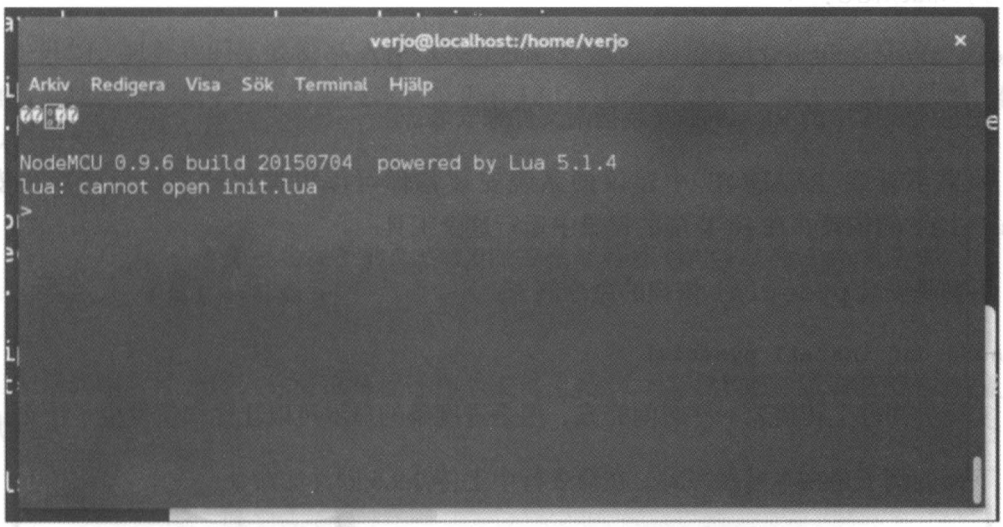
很多地方都可以买到 NodeMCU Amica，从电子商店到网上经销商。

买到 NodeMCU 不难，从硬件的角度来说项目也很简单，在实践中也可以采用 Arduino 或者树莓派，如果它们更容易获得。

下面是一些 NodeMCU 的入门提示：

- NodeMCU 包含的固件提供了交互式 Lua 解释器，可以通过串行端口访问。你通过串行线来直接开发代码。在你的开发机上安装串行通信软件。这样的软件有很多，比如在 Linux 下的 Minicom 和 Windows 下的 Putty。
- 使用串口设置 9600 波特率、八位、无奇偶校验和一个停止位。这个设置通常缩写为 9600 8N1。
- 既然我们已经有了串行终端连接，将 NodeMCU 连到 USB 端口，切换到终端，验证你在终端的窗口看到了提示符。

如果你使用的是 Minicom，提示的窗口如下：



开始写代码前，根据具体的 NodeMCU 的出厂设置，你可能需要往设备烧录固件的镜像。如果在前一步你看到了提示符，就不需要烧录固件的镜像。如果以后你需要在镜像中加更多的特性就得重新烧录镜像。

如果需要烧录一个新的固件镜像，首先从固件源仓库发布的链接。发布的版本地址在这里：`https://github.com/nodemcu/nodemcu-firmware/releases`。

下面是一个用 `wget` 命令下载固件的例子。发布的版本号用整数和浮点数表示，根据

你的需要去选择具体的版本。对于嵌入式的应用来说，整数的固件版本通常就足够用了：

```
wget https://github.com/nodemcu/nodemcu-firmware/releases/
download/0.9.6-dev_20150704/nodemcu_integer_0.9.6-dev_20150704.bin
```

你也可以在开发机器上通过 GitHub 源码直接构建固件镜像，或者也可以根据你的规格使用在线构建服务区构建一个固件。

在线构建的服务地址是 `http://nodemcu-build.com/`。值得一看。如果不出意外，构建统计图颇为耐人寻味。

既然已经有了一个合适的固件文件，你需要安装固件烧录工具，才能把固件镜像文件上传到 NodeMCU：

```
git clone https://github.com/themadinventor/esptool.git
```

按照代码库中的 README 安装指南文件来安装。

如果不喜欢 README 中建议的系统安装，你可以根据你的系统发行版去安装 `pyserial` 的依赖并在 git 克隆的目录中运行这个工具。

下面是安装 `pyserial` 依赖的命令例子：

```
sudo dnf install pyserial
```

实际的固件上传需要一些时间完成，但是进度条的显示可以让你知道发生了什么。

下面的例子是在本书撰写时，在命令行中上传 0.9.6 固件的命令：

```
sudo python ./esptool.py --port /dev/ttyUSB0 write_flash 0x00000 nodemcu_
integer_0.9.6-dev_20150704.bin
```

如果在连接 NodeMCU 时串行命令行出现乱码，你可能需要为烧录固件的命令提供一些额外的参数：

```
sudo esptool.py --port=/dev/ttyUSB0 write_flash 0x0 nodemcu_
integer_0.9.6-dev_20150704.bin  -fs 32m -fm dio -ff 40m
```

命令 `esptool` 也有其他的功能，可以用来验证设置：

```
sudo ./esptool.py read_mac
```

```
Connecting...
MAC: 18:fe:34:00:d7:21
sudo ./esptool.py flash_id
Connecting...
Manufacturer: e0
Device: 4016
```

固件上传完成后，重置 NodeMCU。

这个时候你应该已经有了一个带有 NodeMCU 欢迎提示的串行终端。通过使用工厂提供的 NodeMCU 固件或者上传一个新的固件到设备都可以达到这个状态。

现在，我们开始试试一些“hello world”风格的练习。

一开始，只要我们连接到 NodeMCU Amica 板 GPIO 引脚 0 上，LED 就开始闪烁。如果你有其他类型的板子，你需要找出它是否有 LED，如果有，输入/输出引脚是哪根。你也可以自己包装一个 LED。

注意一些板子的变种 LED 应该连接在 GPIO 引脚 3 而不是这里假定的引脚 0。

如果终端软件允许，你可以将程序作为文件上传到 NodeMCU，或者直接在终端上敲击代码。

NodeMCU 类库的文档在：http://www.nodemcu.com/docs/，它提供了功能使用的很多例子。

你可以首先试着点亮 LED：

```
gpio.write(0, gpio.LOW)  -- turn led on
```

然后用下面的命令关闭 LED：

```
gpio.write(0, gpio.HIGH) -- turn led off
```

现在，你可以循环下面的语句，其中穿插一些延迟语句：

```
while 1 do                          -- loop forever
     gpio.write(0, gpio.HIGH)       -- turn led off
     tmr.delay(1000000)             -- wait one second
     gpio.write(0, gpio.LOW)        -- turn led on
     tmr.delay(1000000)             -- wait one second
end
```

此时，你应该能够验证一个基本工作的设置。直接在终端输入代码有点原始。

NodeMCU 有不同的开发环境来提高开发的体验。

我对 Emacs 编辑器情有独钟，并使用了 NodeMCU Emacs 的模式。NodeMCU 模式可以从 Github 下载。Emacs 发起串行连接的内置功能。当然读者应该使用他/她觉得最舒服的环境。

在能够完成实验前，我们需要一些额外的提示。使用以下命令连接到无线网络：

```
wifi.setmode(wifi.STATION)
wifi.sta.config("SSID","password")
```

SSID 和 password 需要用网络真实的 SSID 和密码替换掉。

如果 NodeMCU 正确连接你的无线网络，这个命令会打印出从网络的 dhcpd 服务器获得的 IP 地址：

```
print(wifi.sta.getip())
```

这段代码会连接到 www.nodemcu.com 的 HTTP 服务器并且打印返回码：

```
conn=net.createConnection(net.TCP, false)
conn:on("receive", function(conn, pl) print(pl) end)
conn:connect(80,"121.41.33.127")
conn:send("GET / HTTP/1.1\r\nHost: www.nodemcu.com\r\n"
   .."Connection: keep-alive\r\nAccept: */*\r\n\r\n")
```

你可能还需要计时功能。下面的代码每隔 1000 毫秒打印 hello world：

```
tmr.alarm(1, 1000, 1, function()
   print("hello world")
```

```
end )
```

在这里，我们声明了一个匿名函数并将其作为参数发送给 `timer` 函数，不经意地显露出了 Lua 的函数型范式。匿名函数每隔 1000 毫秒，也就是 1 秒被调用一次。

要停止 timer，只需要执行：

```
tmr.stop(1)
```

现在，你应该明白了所有可以自行完成实验的细节。如果遇到问题，可以参考本书源代码包中的代码。玩得开心！

总结

在最后一章里，我们学习了新兴的物联网以及它如何影响 DevOps。除了物联网概览以外，我们还制作了一个连接到构建服务器并呈现构建状态的硬件设备。

从抽象到具体的实例，再回到抽象的想法，是贯穿本书的主题。

第 1 章，DevOps 和持续交付简介，我们了解了 DevOps 的背景以及它在敏捷开发世界中的起源。

第 2 章，洞察全局，我们学习了持续交付流水线的不同方面。

第 3 章，DevOps 如何影响架构，深入研究了软件架构领域以及 DevOps 可能对它的影响。

第 4 章，一切皆代码，我们探索了一个发展中的企业如何选择处理它的核心资源源码。

第 5 章，构建代码，介绍了构建系统的概念，比如 Make 和 Jenkins。我们探索它们在持续交付流水中的角色。

在代码构建完成后，我们需要测试它。这对于执行的有效性、无故障发布很重要，我们还看了下第 6 章，测试代码中的一些可用的测试选项。

第 7 章，部署代码，我们探索最终部署我们构建和测试代码到服务器的很多可用选择。

当有代码在运行时，我们需要保证它一直运行。第 8 章，监控代码，考察了我们能保

证代码顺利运行的方式。

第 9 章，问题跟踪，介绍了一些不同的问题跟踪器，这些工具可以帮助我们处理发展流跟踪的复杂性问题。

这是本书的最后一章，漫长的旅程终于结束了。

我希望你和我一样喜欢这段旅行，祝你在 DevOps 的广袤领域中探索成功。

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

DevOps实践

DevOps是一个实践的领域，关注于尽可能高效地交付商业价值。DevOps包含从测试环境直至产品环境的代码要经历的全部流程。它强调不同的角色之间共同协作，以及如何工作得更加紧密，就像这个词语的词根暗示的那样——开发和运维。

在快速地介绍了DevOps和持续交付之后，我们迅速前进到DevOps如何影响架构。你将会创建一个企业级Java应用样例，并将在接下来的章节里工作于其上。通过这种方式，我们探索不同的代码存储和构建服务器。你还将学习使用一些工具来测试代码并成功部署测试环境。接下来，你会学习如何监控代码异常并确保它的正常运行。最后，你还会了解到如何处理日志并保持对问题的跟踪。

本书的目标读者

本书面向愿意承担更大责任，并了解现代企业运转的基础设施的开发人员和系统管理员。本书也很适合愿意更好地支持开发人员的运维人员。你无需事先了解任何的DevOps知识。

通过本书你能学到

- ◎理解DevOps和持续交付的本质并看到DevOps如何支持敏捷流程
- ◎了解系统如何相互匹配并组成一个更大的整体
- ◎创建并熟悉让DevOps更有效率的工具
- ◎用DevOps的思想设计一个适合持续部署的系统
- ◎用诸如Git、Gerrit和GitLab等不同方式高效地存储和管理代码
- ◎配置一个构建CRUD应用样例的任务
- ◎通过Jenkins和Selenium使用自动化回归测试来测试代码
- ◎使用诸如Puppet、Ansible、PalletOps、Chef和Vagrant等工具部署代码
- ◎使用Nagios、Munin和Graphite监控代码健康
- ◎探索Trac的工作方式——一个用于问题跟踪的工具

[PACKT] open source*
PUBLISHING
community experience distilled

责任编辑：张春雨
封面设计：李　玲

上架建议：开发/运维

ISBN 978-7-121-29812-7
9 787121 298127 >

定价：69.00元